1986

Design and simulation of an ultra reliable fault tolerant computing system voter and interstage.

Spurlock, Virgil K.

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DESIGN AND SIMULATION OF AN ULTRA RELIABLE FAULT TOLERANT COMPUTING SYSTEM VOTER AND INTERSTAGE

by

Virgil K. Spurlock

March 1986

Thesis Advisor: L. W. Abbott

ECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
. PERFORMING ORGANIZATION REPORT NUMBER(S)	S. MONITORING ORGANIZATION REPORT NUMBER(S)
a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (If applicable) 62	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
c. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
a. NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
c. ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

1 TITLE (Include Security Classification)
DESIGN AND SIMULATION OF AN ULTRA RELIABLE FAULT TOLERANT COMPUTING SYSTEM VOTER AND INTERSTAGE

2 PERSONAL AUTHOR(S)
Virgil K. Spurlock

3a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM _____ TO _____	86 March	185

6 SUPPLEMENTARY NOTATION

7 COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Fault Tolerant Computing

9 ABSTRACT (Continue on reverse if necessary and identify by block number)

The purpose of this thesis was to design a portion of the hardcore for an ultra reliable fault tolerant computing network. The design focused on the interstage, the midvalue voter, and the interface to the CPU. The design also investigated the use of the custom slave processor mode of the National Semiconductor 32016-10 CPU as the interface to the interstage. The primary focus of the design was reliability. Therefore the number of gates used was minimized as much as possible. Finally, the entire design was constructed and tested on the Valid Logic Inc. SCALD system computer aided design (CAD) workstation. Effectiveness of the CAD system for large designs was also studied.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
X UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Prof L. W. Abbott	22b TELEPHONE (Include Area Code) (408)646-2379 22c OFFICE SYMBOL 62At

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

Approved for Public Release; Distribution is Unlimited

Design and Simulation of an Ultra Reliable Fault Tolerant Computing System Voter and Interstage

by

Virgil K. Spurlock
Captain, United States Army
B.S.E.E., University of Kentucky, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 1986

ABSTRACT

The purpose of this thesis was to design a portion of the hardcore for an ultra reliable fault tolerant computing network. The design focused on the interstage, the mid-value voter, and the interface to the CPU. The design also investigated the use of the custom slave processor mode of the National Semiconductor 32016-10 CPU as the interface to the interstage. The primary focus of the design was reliability. Therefore the number of gates used was minimized as much as possible. Finally, the entire design was constructed and tested on the Valid Logic Inc. SCALD system computer aided design (CAD) workstation. Effectiveness of the CAD system for large designs was also studied.

TABLE OF CONTENTS

I. INTRODUCTION AND OVERVIEW

Increasingly computers are being used to monitor and operate systems utilizing digital controls in which a failure (actuator, sensor, bus, or computer) can have catastrophic and sometimes life threatening results. Failures generally have two sources: 1) random failures such as manufacturing defects and normal wear-out and 2) physical damage such as vibration and battle damage. Fault tolerant computing provides the systems that recognize and adapt to these failures.

Several examples emphasize the need for fault tolerant computing. Studies [Ref. 1] have shown that commercial passenger jets can carry a larger payload and be more fuel efficient if the structures are aerodynamically redesigned to reduce drag. However, the new structures also reduce or eliminate the inherent static stability of the airframe. Computers and digital controls are required to restore stability. A computer failure could leave a pilot with an airplane that cannot be controlled.

In military aviation, computers control a myriad of operations in fighter aircraft that overwhelm a pilot's capabilities. Computer failure can degrade vital flight characteristics, target acquisition, and fire control systems. Physical battle damage to the aircraft increases

the overall probability of computer failure. Computer failure can mean degradation or loss of mission capability.

Many of the attributes required by digital computers in aircraft extend to spacecraft. Rapid and precise orbital calculations, control of precision thruster and booster firings for accurate orbital injection and attitude control, and monitoring of life support systems are a few examples. Again, failure of the computer system can be catastrophic.

The goal of fault tolerant computing systems is to prevent a single point failure from disabling an entire system. Fault tolerant computing systems provide redundant components and an interconnection network that connects redundant fault tolerant computers with an array of actuators and sensors. Fault tolerant design also provides the capability to identify, isolate, and remove a bad component from the system in a transparent and non-disruptive manner.

Systems can be made more reliable by adding redundant sensors, actuators, data links and computers. The fault tolerant system must monitor these redundant components, decide if a failure has occurred, switch to another component, and periodically check the failed component to see if it has somehow become operational again.

A tremendous amount of research has been conducted in fault tolerant computing. Of particular concern to this

thesis is the work conducted at the NASA Ames Research Center, Dryden Flight Reseach Facility on the Dispersed Sensor Processor Mesh (DSPM) [Refs. 2,3] and the research [Ref. 4] of Professor Larry Abbott (Naval Postgraduate School) on an ultra-reliable fault tolerant network.

One fault tolerant network, a basic hybrid redundancy (BHR) organization proposed by Siewiorek [Ref. 5], uses five as the optimal number of computers with three active and two spare computers (Figure 1-1). The rotary multiplexer activates three computers, the other two being spare or failed. Through the multiplexer, three separate data streams are sent to an voter which rejects a value when it does not match the other two. This basic hybrid redundancy system requires lock-step synchronization and a bit by bit comparison of data. The voter and multiplexer must be as simple and reliable as possible since a failure in either brings down the entire system.

The BHR, with identity comparisons and lock-step synchronization cannot address unsynchronized external clocks or N-version programming. N-version programming is a software technique in which different languages or algorithms compute the same function. With only one algorithm, a software error and it's resulting output error could go undetected. Therefore, three separate algorithms, each working the same problem should produce three identical answers. This is not always the case as the three answers

may vary slightly due to the different algorithms (i.e., in the last few significant digits) but the differences are insignificant. BHR identity comparison would immediately reject the differing outputs whereas an SIR mid-value voter would not.

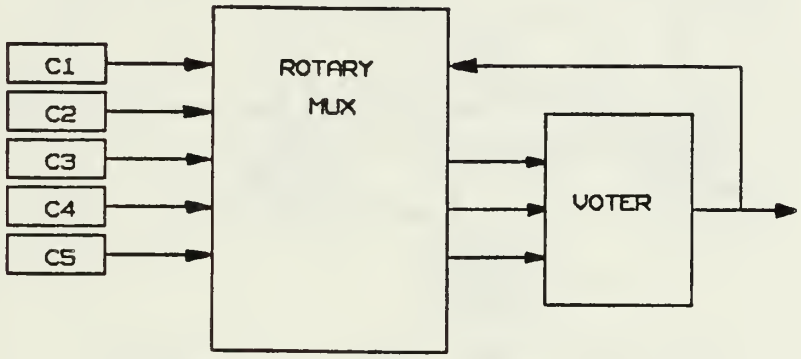


Figure 1-1: Basic Hybrid Redundancy

Synergistically integrated reliability (SIR) is an advanced hybrid redundancy scheme shown in Figure 1-2. The SIR architecture transmits data to the DSPM (based on previous work by Smith [Ref. 6]), an external communications circular network that monitors and controls the buses providing data to and from the SIR and the sensors/actuators.

The SIR combines a number of reliability methods to achieve hardware and software reliablity. The reliability methods include hybrid redundancy, N-version programming, source congruent data interchanges, and hybrid redundancy management [Ref. 4].

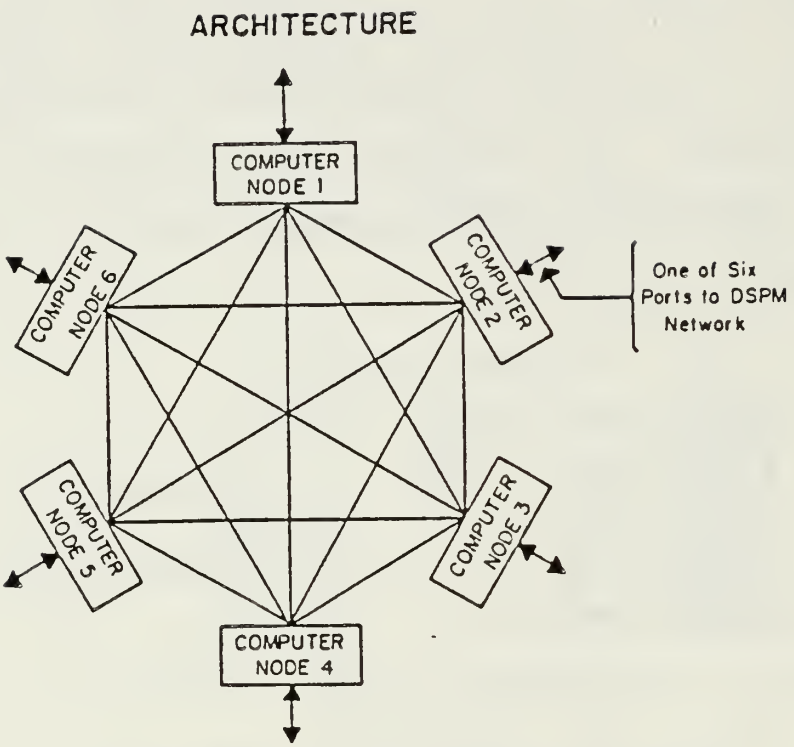


Figure 1-2: SIR Architecture

The hardcore of the SIR node (Figure 1-3) consists of a mid-value voter, an interstage, and a rotary multiplexer. Each SIR node has a computer and a hardcore. Each node communicates to the five identical SIR nodes.

The rotary multiplexer simply channels data from the node computer to the interstage and from the interstage to the external computers. The multiplexer must handle full duplex communications which increases its complexity compared to the BHR, but this is more than offset since multiplex routing is handled in software by the SIR concept. Overall, the SIR multiplexer is significantly less complex than the BHR multiplexer by approximately 2:1 [Ref. 4].

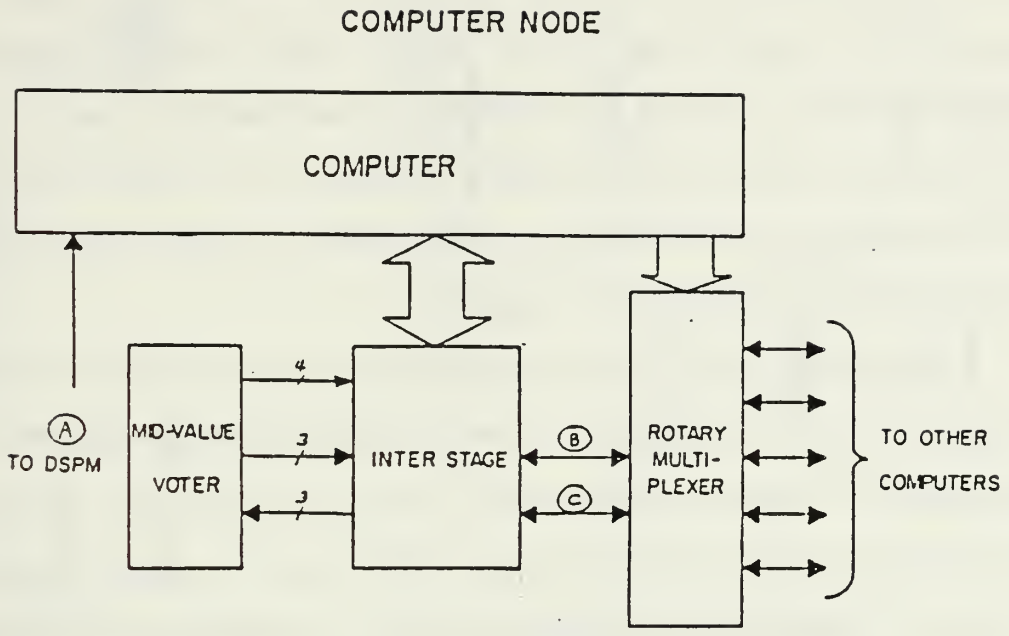


Figure 1-3: SIR computer node

The voter in the SIR computes a middle value from the three data values stored in the interstage. Two values are provided by the rotary multiplexer from external interstages and the third, from the DSPM, is provided through the CPU. Because N-version programming can provide outputs whose relative differences are insignificant, the mid-value voter output in the SIR concept can still decide a "correct" value whereas the identify comparison of BHR (which rejects differing values) cannot. SIR also supports data congruency identity comparisons.

The interstage provides full duplex data transfer as well as data recirculation to provide data congruency. The interstage can receive a single copy of data, triplicate it,

and recirculate it to the other computers. The voter serially receives data from the interstage and simultaneously serially transmits the results back. Voted results and values can be sent to the CPU or the results can be routed to the external computers via two different channels for redundancy checks.

This thesis is concerned with the hardware design of the voter, the interstage, and interfacing the system to the NS32Ø16-1Ø CPU (Figure 1-4). Simplicity in the design is an overriding concern since reliability is proportional to the gate count. The secondary goal of the thesis will be to use and evaluate the Valid Logic Inc. CAD/CAM system for validation and simulation of the design.

A. VALID INC. SCALD COMPUTER AIDED DESIGN (CAD) SYSTEM

The interstage, voter, and CPU interface was designed and tested using the Valid Inc. Scald CAD system. Use of CAD tools can validate a design before breadboard construction. Tracing errors on a breadboard can be a very difficult, frustrating, and time consuming process. The VALID/Scald system is used to verify the design and should significantly reduce the inevitable troubleshooting required during construction. Discovery of timing errors is expected to be the most visible result.

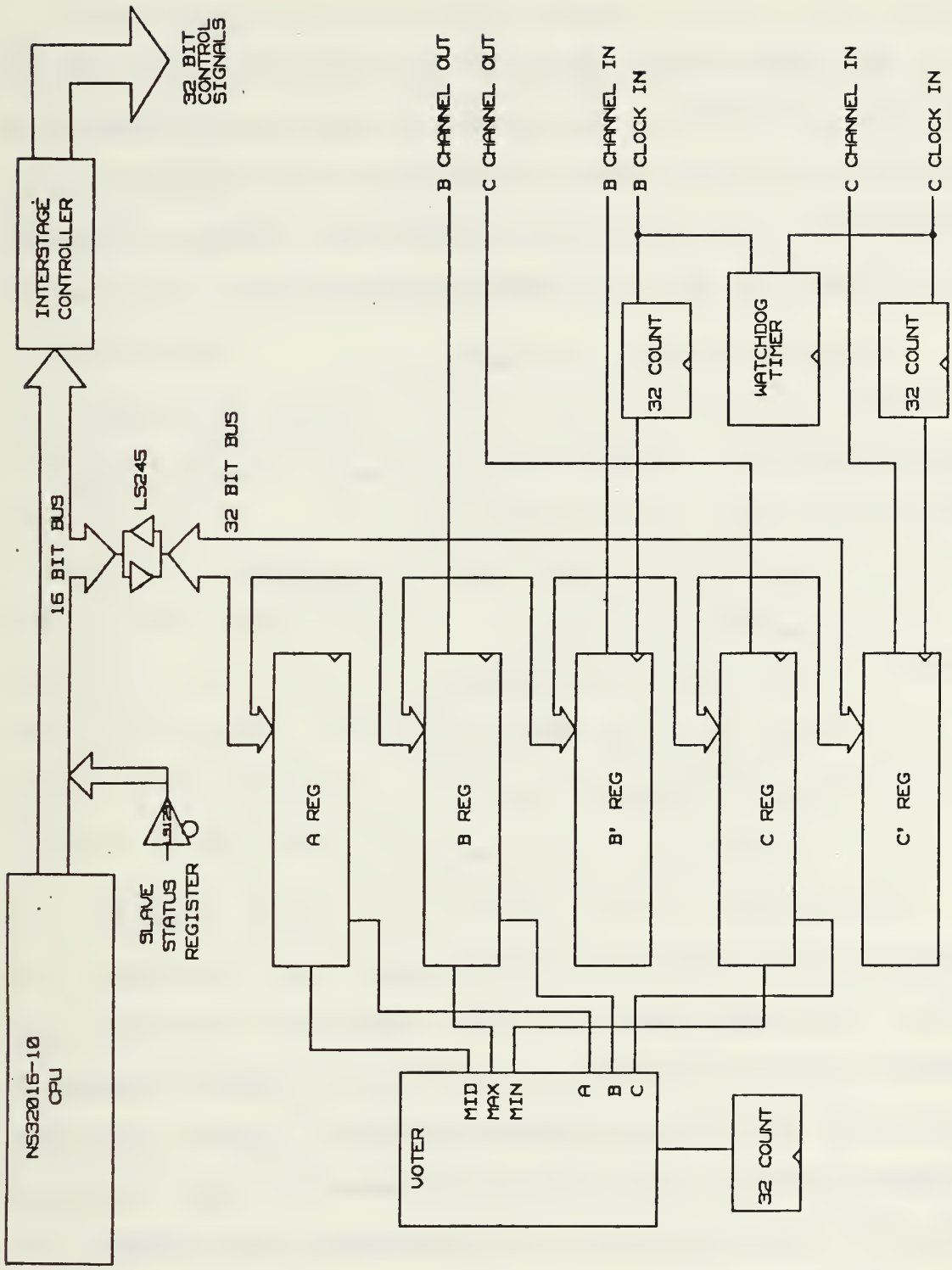


Figure 1-4: Interstage and NS32016-10 CPU

The experimental prototype will be built from the CAD design. Therefore, schematics must be as accurate as possible and the bugs and errors worked out prior to construction of a prototype. Appendix A explains the Scald CAD terminology used throughout this thesis.

B. NS32016-10 CENTRAL PROCESSING UNIT

The NS32016-10 CPU is a member of National Semiconductor's 32000™ microprocessor family [Ref 7]. The CPU has the following features:

- Supports a Custom Slave Processor
- Operates at 10 Mhz
- True 32-bit architecture
- High-speed XMOS™ Technology
- 16-bit external bus
- 1.5 Watt power dissipation

The NS32201 Timing Control Unit (TCU) provides the required two phase non-overlapping clock pulses. The NS32016-10 also supports three slave processors: the NS32082 Memory Management Unit (MMU); the NS32081 Floating-Point Unit; and a Custom Slave Processor. Figure 1-5 shows a typical system interconnection diagram for the NS 32016-10 CPU. Although the Custom Slave Processor is not shown, its connections to the CPU are identical to the NS32081 floating point unit.

The Custom Slave Processor mode is used in the fault tolerant computing system under design. It appears that using the Custom Slave Processor mode will be faster than using a conventional peripheral, but will require certain tradeoffs between performance and reliability. Suitability of the Custom Slave Processor mode will also be investigated in this thesis.

Custom designed by the user, the Custom Slave Processor interfaces to the CPU through predefined instruction sets and protocols. The instruction sets are an excellent method for passing instructions, statuses and results between the two chips. The rigidly defined instruction protocol also reduces the amount of hardware in the Custom Slave Processor. The interstage will be designed as a Custom Slave Processor. Details are provided in Section II, "Custom Slave Interface".

C. VOTER

The voter is a three way serial comparator Mealy finite state machine. The first bit of three numbers (A, B, and C) is serially input to the voter. The voter decides which of the three is the maximum, middle, or minimum value then serially outputs the respective bits into shift registers.

Serial voting in this manner requires 13 states. Figure 1-6 shows the state diagram. Since 4 bits are required for the states and there are three inputs and

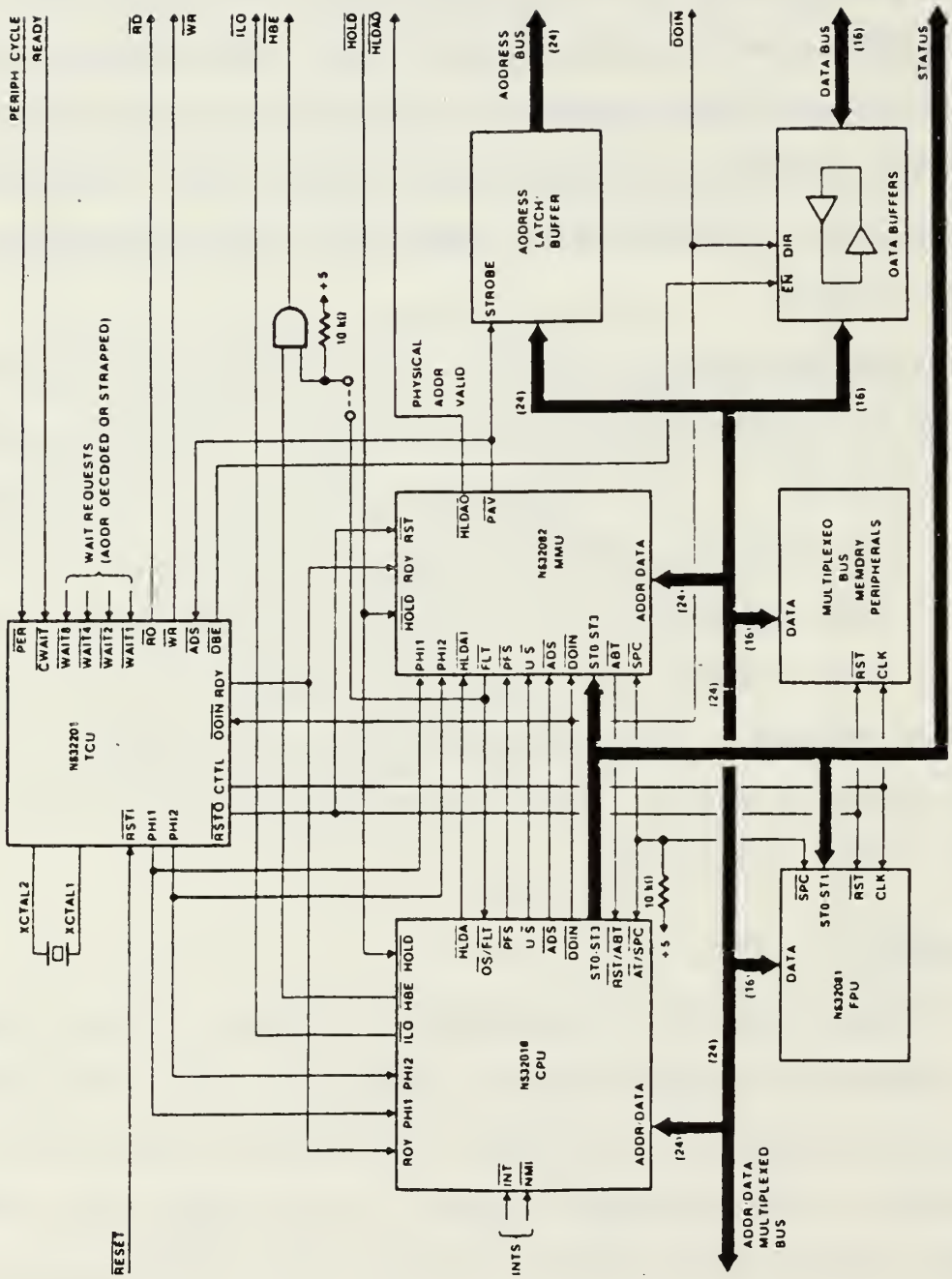


Figure 1-5: System Interconnection Diagram

outputs, this is a seven variable problem, too large to use a Karnaugh map for minterm reduction. A Quine-McCluskey minterm reduction computer algorithm is required and an algorithm was written by the author based on the techniques outlined by Mano [Ref. 8]. The algorithm, written in BASIC, is in Appendix B. Details of the voter design are in Section VI, "Voter".

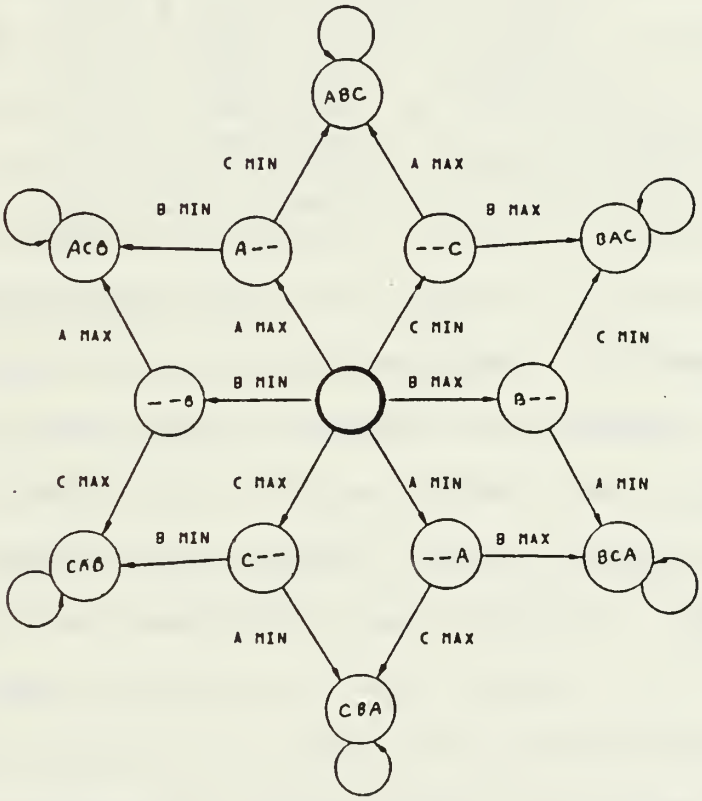


Figure 1-6: Voter State Diagram

D. INTERSTAGE

The interstage is the heart of SIR the fault tolerant system. It stores data from external interstages until needed by the voter or the CPU. The interstage can

recirculate the stored data and transmit it to the external interstages. The interstage is the direct link between the CPU and the external network that connects all the CPU's.

The interstage receives three 32 bit numbers, one from the CPU via the 16 bit data bus, the other two serially from external interstages via channels B and C (refer to Figure 1-4). The 32-bit value from the CPU is parallel loaded into the A register. The two serial inputs use a 10 MHz clock but are not synchronized to each other or to the CPU. The two values from channels B and C are initially serially loaded into the B' and C' registers respectively and eventually parallel loaded into the B and C registers (by the VOTE command). The B' and C' registers are now able to receive and store new data while the previous data is manipulated. Use of the B, B', C, and C' 32-bit registers allows full duplex serial communications with the other interstages. Details are provided in Section IV, "Interstage".

The interstage is designed to respond to only eight commands, a constraint resulting from minimizing hardware in the interstage while using the Custom Slave Processor mode. The interstage can perform all necessary functions using only eight commands. The cost of additional hardware in the controller to decode and respond to more instructions could not be justified since additional instructions are not

necessary. Details on how the instruction set was chosen is provided in Section III, "Interstage Instruction Set". The eight instructions for the interstage are:

1) Load WDTREG; INT or FP values
2) Load 32 bit value from CPU to A register
3) Vote
4) Load 32 bit value from A register to CPU
5) Load B register to A register
6) Load C register to A register
7) Load A register to B & C registers
8) Serially transfer B & C register data out

E. INTERSTAGE CONTROLLER

The controller interfaces between the interstage and the CPU. The controller receives and decodes instructions and operands from the CPU then issues control signals to the interstage and voter. Upon completion of an instruction, the controller signals the CPU, passes data (if necessary) then switches to a wait state.

Timing between the CPU, the controller, and the interstage is critical. The 10 MHz CPU passes information to the slave during two clock cycles, T1 and T4, (200 ns period). The interstage, on the other hand, operates directly at 10 MHz (100 ns period). Synchronization between the two is provided by the controller. Details are provided in Section V, "Interstage Controller".

II. CUSTOM SLAVE INTERFACE

The interstage interfaces with the NS 32016-10 CPU [Ref. 7] by operating as a custom slave processor. The CPU recognizes three slave processor instruction sets: floating point instructions, memory management instructions, and the custom slave instructions. The CPU's 4-bit configuration register (CFG) detects the presence of specific external devices and is accessed through one command, SETCFG, which is set by system initialization after initial startup or RESET. The "C" bit in the CFG must be set or else the slave instructions will trap as undefined. A primary focus of this thesis is the use of the interstage as an external device to the CPU by use of the custom slave processor mode.

There are only three interfaces between the CPU and the Custom Slave Processor (Figure 2-1). They are the 16-bit data bus (D<15..0>), a 4 bit CPU cycle status line (ST<3..0>), and a bi-directional data strobe called the Slave Processor Control (SPC*). Clock and RESET signals are generated external to the CPU and Custom Slave Processor.

Each instruction has a three byte field. The first byte is an identification byte (ID) and the next two bytes constitute an operation word. The ID byte is placed onto the lower 8 bits of the data bus and has three functions:

1) Identifies the instruction as being for a slave processor.

2) Specifies which Slave Processor will execute it.

3) Specifies the format of the instruction word.

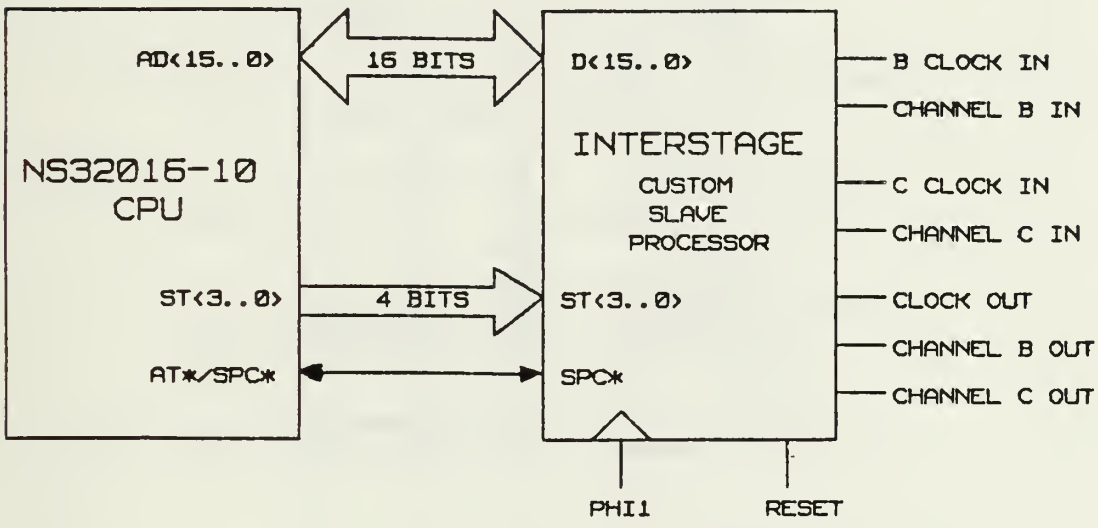
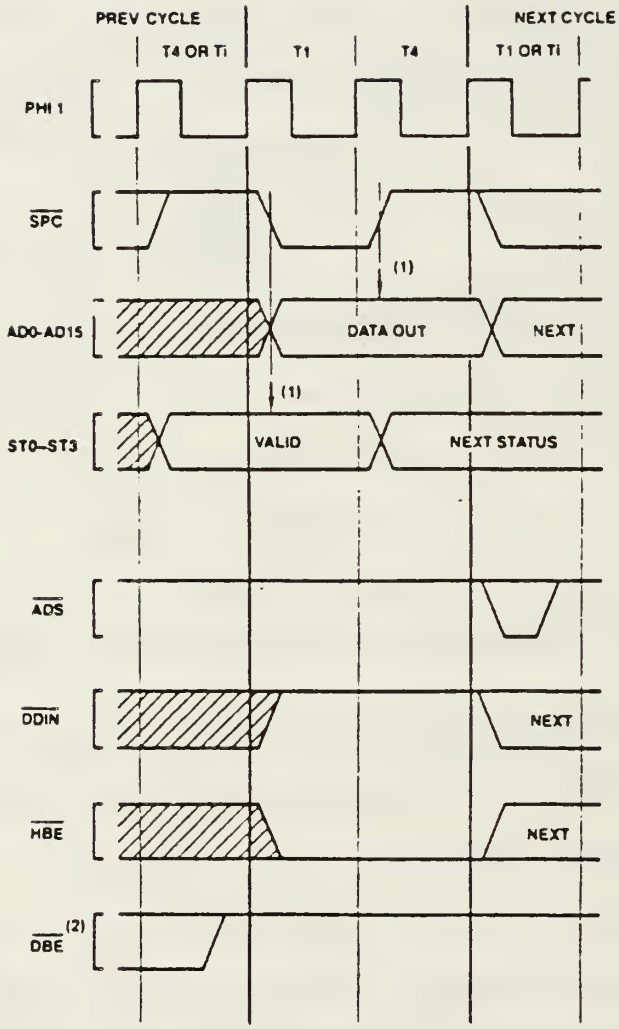


Figure 2-1: CPU and Custom Slave Processor Interface

The Slave Processor bus cycle always takes exactly two clock periods. The two intervals are labeled T1 and T4. SPC* goes low during T1 and the Slave Processor latches the status from ST<3..0>. SPC* then goes high during T4 and data on the CPU bus is latched on the leading edge of SPC*. Figure 2-2 shows the CPU to Slave Processor Write cycle. Data is read to the CPU from the Slave Processor on the leading edge of SPC* (See Figure 2-3).

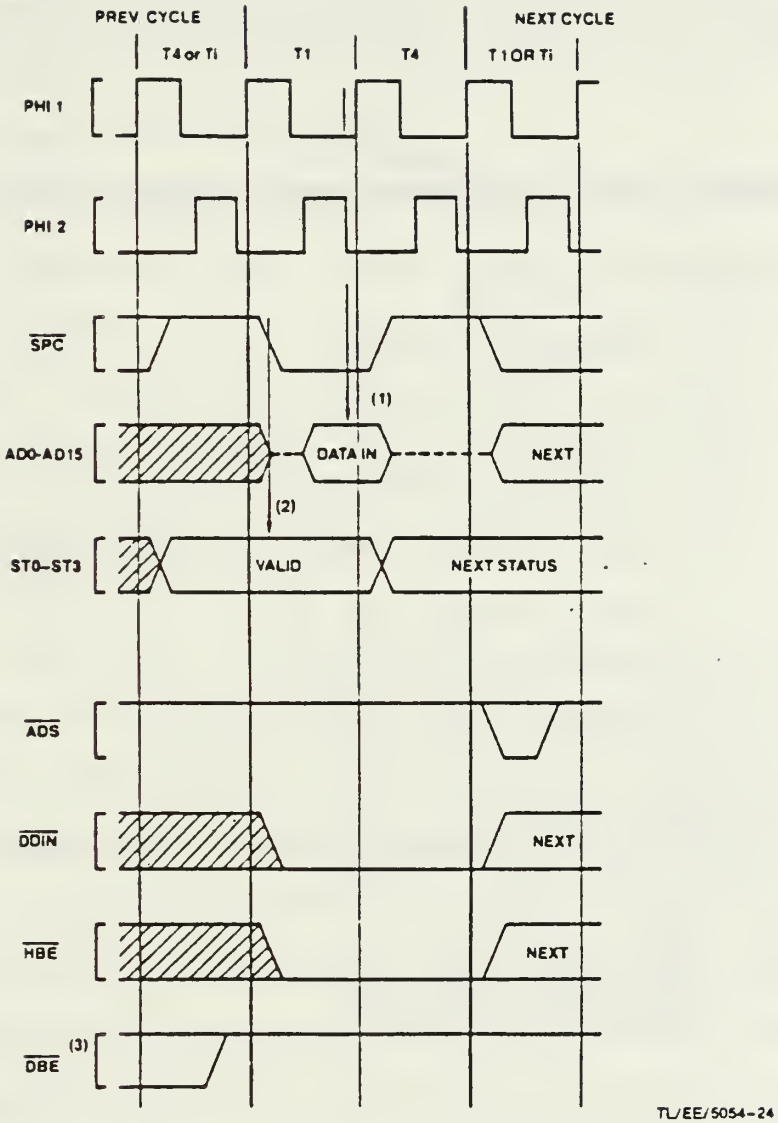


Note:

(1) Arrows indicate points at which the Slave Processor samples.

(2) $\overline{DBE}$, being provided by the NS32201 TCU, remains inactive due to the fact that no pulse is presented on $\overline{ADS}$. $\overline{TCU}$ signals $\overline{RD}$, $\overline{WR}$ and $\overline{TSO}$ also remain inactive.

Figure 2-2: CPU Write to Custom Slave Processor



Note:

(1) CPU samples Data Bus here.

(2) Slave Processor samples CPU Status here.

(3) $\overline{DBE}$ and all other NS32201 TCU bus signals remain inactive because no $\overline{ADS}$ pulse is received from the CPU.

Figure 2-3: CPU Read from Custom Slave Processor

The Custom Slave Processor continuously monitors the 4-bit ST<3..Ø> lines. When the Slave bit is set in the CFG register and the CPU receives a Custom Slave Processor instruction, it initiates the sequence shown in Table 2-1.

TABLE 2-1
CUSTOM SLAVE PROCESSOR COMMUNICATIONS PROTOCOL

Step	ST<3..Ø>	ACTION
1	ID	CPU Sends ID Byte
2	OP	CPU Sends Operation Word
3	OP	CPU Sends Operands (B, W, D, Q)
4	-	Slave Starts Execution, CPU Pre-Fetches
5	-	Slave Pulses SPC* Low
6	ST	CPU Reads Status Word
7	OP	CPU Reads Results (if any)

Broadcast ID (ID): 1111
Transfer (Read/Write) Operand (OP): 11Ø1
Read Status (ST): 111Ø

B - byte (8 bits)
W - word (16 bits)
D - doubleword (32 bits)
Q - quadword (64 bits)

In Step 1, Table 2-1, the Broadcast ID (1111) is sent over ST<3..Ø> and the ID byte is transferred on the least significant byte of the data bus (D<7..Ø>). All slave processors receive and decode this byte but only the slave

processor selected is activated. The ID byte for the Custom Slave Processor is:

nnn1Ø11Ø

where the "nnn" denotes a particular Operation Word Format. After sending this ID byte, the CPU is communicating only with the Custom Slave Processor.

Table 2-2 shows the Custom Slave Instruction Protocols. Figure 2-4 shows that the Custom Slave Processor has three distinct formats: Format 15.Ø (nnn = ØØØ); Format 15.1 (nnn = ØØ1); Format 15.5 (nnn = 1Ø1). A total of twenty instructions are available if all three formats are used (Format 15.Ø provides four and Format's 15.1 and 15.5 provides eight each).

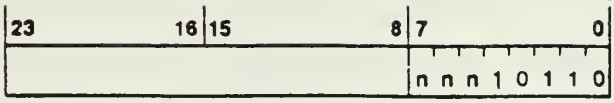
When using the Custom Slave Processor instruction set, the designer builds the custom slave processor to interpret the OP Code fields and the types of data transferred (byte, word, doubleword, or quadword) to suit the design. Referring to Table 2-2, the command CCVØci has two operands: the first is a value to be read from the CPU and transferred to the Custom Slave Processor; the second is the value to be written to the CPU from the Custom Slave Processor. The first operand is labeled "c" which means the operand can be either a doubleword or a quadword. Operand 2, the returned value, is an "i" which means the operand returned to the CPU can be either a byte, word, or doubleword.

TABLE 2-2
CUSTOM SLAVE INSTRUCTION PROTOCOLS

Mnemonic	Operand 1 Class	Operand 2 Class	Operand 1 Issued	Operand 2 Issued	Returned Value Type and Dest.	PSR Bits Affected
CCAL0c	read.c	rmw.c	c	c	c to Op. 2	none
CCAL1c	read.c	rmw.c	c	c	c to Op. 2	none
CCAL2c	read.c	rmw.c	c	c	c to Op. 2	none
CCAL3c	read.c	rmw.c	c	c	c to Op. 2	none
CMOV0c	read.c	write.c	c	N/A	c to Op. 2	none
CMOV1c	read.c	write.c	c	N/A	c to Op. 2	none
CMOV2c	read.c	write.c	c	N/A	c to Op. 2	none
CCMPc	read.c	read.c	c	c	N/A	N,Z,L
CCV0ci	read.c	write.i	c	N/A	i to Op. 2	none
CCV1ci	read.c	write.i	c	N/A	i to Op. 2	none
CCV2ci	read.c	write.i	c	N/A	i to Op. 2	none
CCV3ic	readi	write.c	i	N/A	c to Op. 2	none
CCV4DQ	read.D	write.Q	D	N/A	Q to Op. 2	none
CCV5QD	read.Q	write.D	Q	N/A	D to Op. 2	none
LCSR	read.D	N/A	D	N/A	N/A	none
SCSR	N/A	write.D	N/A	N/A	D to Op. 2	none
CATST0*	addr	N/A	D	N/A	N/A	F
CATST1*	addr	N/A	D	N/A	N/A	F
LCR*	read.D	N/A	D	N/A	N/A	none
SCR*	write.D	N/A	N/A	N/A	D to Op.1	none

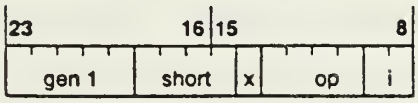
NOTE:
D = Double Word
i = integer size (B,W,D) specified in mnemonic.
f = Floating,Point type (F,L) specified in mnemonic.
N/A = Not Applicable to this instruction.

CCV5QD does not give the user any options on the data types. It reads a quadword from Operand 1 and writes it to the Custom Slave processor. Upon completion of the instruction by the Custom Slave Processor, a doubleword is written to the CPU and stored in the Operand 2 memory address. The instruction LCSR writes the doubleword in Operand 1 to the Slave but unlike the previous two, does not read a value back to the CPU.



Format 15
(Custom Slave)

nnn **Operation Word Format**

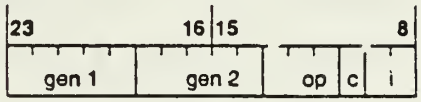


000

Format 15.0

CATST0	−0000	LCR	−0010
CATST1	−0001	SCR	−0011

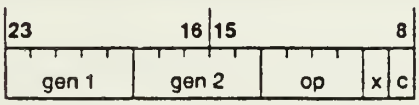
Trap (UND) on all others



001

Format 15.1

CCV3	−000	CCV2	−100
LCSR	−001	CCV1	−101
CCV5	−010	SCSR	−110
CCV4	−011	CCV0	−111



101

Format 15.5

CCAL0	−0000	CCAL3	−1000
CMOV0	−0001	Trap (UND)	−1010
CCMP	−0010	Trap (UND)	−1011
CCAL1	−0100	CCAL2	−1100
CMOV2	−0101	CMOV1	−1101
Trap (UND)	−0110	Trap (UND)	−1110
Trap (UND)	−0111	Trap (UND)	−1111

If nnn = 010, 011, 100, 110, 111
then Trap (UND) Always

Note: Interstage uses Format 15.1

Figure 2-4: Custom Slave Instruction Formats

In Step 2, Table 2-1, the CPU sends the operation word while applying ST<3..Ø> = 11Ø1, "Transfer (Read/Write) Slave Operand". After decoding the operation word, the Custom Slave Processor knows the instruction and the size and number of operands to be transferred. Internal engineering considerations in the NS32Ø16-1Ø cause the bytes on the data bus to be swapped. That is, the lower byte, D<7..Ø>, appears on pins AD<15..8> and the higher byte, D<15..8>, appears on pins AD<7..Ø>.

Status code 11Ø1 remains on ST<3..Ø> while the CPU fetches the operands and transfers them to the Custom Slave Processor. Since the Custom Slave Processor determines the size of the operands sent by decoding the operation word, it can begin execution as soon as the data has been received. The CPU then holds AT*/SPC* high through a 5K pullup device, prefetches the next instruction and waits for the Slave Processor to signal completion by pulsing SPC* low.

The 74LS245 octal bus transceiver with tri-state outputs (see Figure 1-2) in the interstage isolate the CPU's bus from the Custom Slave Processor's bus. Isolation is very important because the CPU prefetches the next instruction concurrent with the Custom Slave Processor instruction execution. If the buses were not isolated, bus conflict could occur.

When the Custom Slave Processor has completed execution of the instruction, it pulses SPC* low during T1. Upon detection of the pulse, the CPU reads the status word at the next T1 while applying status code 1110 "Read Slave Status" on ST<3..0>. It is imperative that the "Quit" bit is not set. If set, this indicates an error was detected by the Slave Processor and the CPU will immediately trap to the interrupt table. The slave processor status word format is shown in Figure 2.5

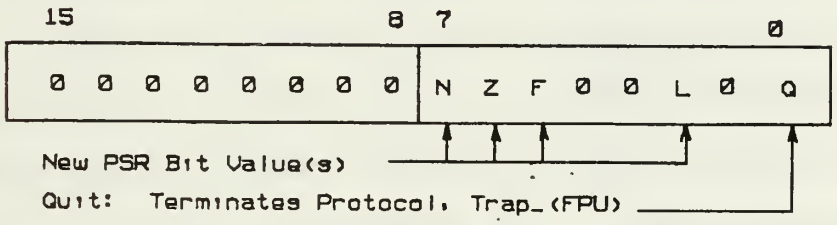


Figure 2-5: Slave Processor Status Word Format

The final step in Table 2-1 reads data from the Slave Processor (if dictated by the instruction) and transfers it to its destination operand address in the CPU. Status code 1101, "Transfer Slave Operand", is applied to ST<3..0> during the read and write process.

Upon completion of the Slave Processor protocol, communications between the CPU and the Custom Slave are broken. The CPU continues program execution independent of the status of the Custom Slave. The Custom Slave remains off the data bus and monitors the ST<3..0> status lines for the next Send ID signal.

III. INTERSTAGE INSTRUCTION SET

The interstage instruction set is a subset of the custom slave processor instruction set built into the NS32Ø16-1Ø CPU. Figure 2-4 shows the custom slave processor instruction formats. Use of all three formats provides twenty instructions: format 15.Ø has four; format 15.1 has eight; and format 15.3 has eight. Only eight instructions are needed. Additional instructions increase the hardware required for decoding and are therefore not used.

The bit ordering of each format is not consistent. In particular the opcode for each format (which can be three or four bits wide) is in different locations within the byte. Since the opcode position and size is not static, extra hardware is required for decoding.

Besides the opcode, there are other bit fields in the 16-bit instruction format. Three separate bit fields determine the size of the transferred operands and are outlined in Table 3-1. The remaining bit fields, "gen1", "gen2", and "short" are used for addressing within the CPU and do not affect the interstage.

Of the three instruction formats available, format 15.1 was chosen for the interstage. Format 15.Ø, with four instructions, was eliminated from consideration. Both format 15.1 and 15.5 have eight instructions each, but the

opcode for format 15.1 is only three bits wide compared to four for format 15.5. This difference saves a register in the interstage and favors format 15.1

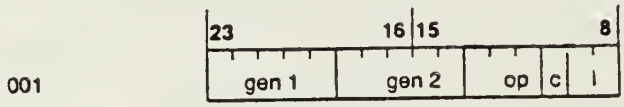
TABLE 3-1
OPERAND SIZES

Mnemonic	Code	Word Size
i	ØØ	Byte
	Ø1	Word
	1Ø	Not Used
	11	Doubleword
c	Ø	Quadword
	1	Doubleword
x	x	Not Used

After every instruction, the CPU reads a status word from the custom slave processor. None of the eight instructions in format 15-1 affect the processor status register. Therefore, the processor status register bits, buffered off the bus, can be held low by grounding. Registers are not needed to store the status bits. For the above reasons, format 15.1 (Figure 3-1) was chosen to provide the instruction set for the interstage.

Decoding hardware is further reduced by keeping the "i" and "c" operand codes constant. The "i" bits are always set at "11", doubleword, and the "i" bit is set for doubleword.

While this may be a little inefficient for the CPU, it does not adversely affect the CPU or the interstage.



Format 15.1

CCV3	-000	CCV2	-100
LCSR	-001	CCV1	-101
CCV5	-010	SCSR	-110
CCV4	-011	CCV0	-111

Mnemonic	Operand 1 Class	Operand 2 Class	Operand 1 Issued	Operand 2 Issued	Returned Value Type and Dest.	PSR Bits Affected
CCV0ci	read.c	write.i	c	N/A	i to Op. 2	none
CCV1ci	read.c	write.i	c	N/A	i to Op. 2	none
CCV2ci	read.c	write.i	c	N/A	i to Op. 2	none
CCV3ic	readi	write.c	i	N/A	c to Op. 2	none
CCV4DQ	read.D	write.Q	D	N/A	Q to Op. 2	none
CCV5QD	read.Q	write.D	Q	N/A	D to Op. 2	none
LCSR	read.D	N/A	D	N/A	N/A	none
SCSR	N/A	write.D	N/A	N/A	D to Op. 2	none

Figure 3-2: Interstage Instruction Format and Protocol

There are several assumptions made in constructing the instruction set for the interstage. First, not all the capabilities provided by the NS32016-10's custom slave processor instruction set are needed. For example, if the instruction sends two doublewords to the interstage across the external 16-bit bus and the interstage only needs one doubleword, the second is ignored by the interstage.

If the CPU is expecting a word to be returned from the interstage and the interstage does not need to pass a value,

the CPU will sample the 16-bit bus and latch a series of high impedance outputs. Garbage will be stored in the operand address, but according to National Semiconductor applications engineering, this will not adversely affect the CPU.

Whoever writes the software for the CPU will have to provide dummy storage space. In certain cases there will be operand addresses transferred to the interstage that the interstage does not require and will ignore. In other cases the CPU receive "garbage" from the interstage that will be stored in operand addresses that should be ignored by the software. The VOTE command is the only instruction that will return an operand to the CPU.

The eight instructions used by the interstage will now be described. Drawn from format 15.1, they satisfy the requirements of the SIR network.

The first instruction, CCCV3ci, "000", is used to pass the instruction to copy the contents of the A register into the B and C registers. The doubleword written to the interstage will be ignored. The doubleword the CPU reads back will be "garbage". The instruction will signal the interstage to transfer data internally only.

The second instruction, LCSR, "001", will command the interstage to serially transfer the contents of the B and C registers to the external interstages. LCSR passes a

doubleword to the interstage which is ignored. The CPU will not read from the interstage.

Instruction three, CCV5QD, "Ø1Ø", will write a quadword to the interstage and read a doubleword back. The interstage is only concerned with reading a 16-bit value to the watchdog timer register and a 1-bit value to indicate whether the vote will be for an integer or an IEEE standard floating point number [Ref. 9]. Although the CPU will expect to read a doubleword at the end of instruction execution, the interstage will not write anything.

The fourth instruction, CCV4DQ, "Ø11" is used to instruct the interstage to vote the three 32-bit values stored in the A, B, and C registers. The doubleword transferred to the interstage is ignored. After completion of the vote, the CPU will read a quadword. The first 32 bits transferred to the CPU will be the middle value of the vote. The next 8 bits contain the slave status register (SSR). The SSR contains the state of the voter (4 bits), the state of the B and C channel timers (2 bits), and the state of the watchdog timer (1 bit). The last bit in the byte is not used. The final 24 bits of the quadword will not contain any information.

The fifth instruction, CCCV2ci, "1ØØ", is used to transfer a 32-bit value from the A register to the CPU. The CPU will write a doubleword to the interstage (which is

promptly ignored). The doubleword read by the CPU will be from the A register.

Instruction six, CCCV1ci, "101", will be used to copy a doubleword from the CPU to the A register in the interstage. The doubleword written to the interstage will be stored in the A register. The doubleword read by the CPU will be meaningless.

Instruction seven, SCSR, "110", is used to pass an instruction to the interstage. The doubleword sent to the interstage is ignored. SCSR does not read a value from the interstage. The interstage uses this instruction to copy the contents of the C register into the A register.

The final instruction, CCCV0ci, "111", is also used to pass an instruction: copy the contents of the B register into the A register. The doublewords transferred and received are mutually ignored.

The interstage instruction set and the custom slave processor instructions they are derived from are shown in Table 3-2.

TABLE 3-2
INTERSTAGE INSTRUCTION SET

OPCODE	CPU INSTRUCTION	INTERSTAGE INSTRUCTION
000	CCCV3ci	A - B,C
001	LCSR	Serial Out
010	CCV5QD	Load WDT Reg, FP/INT Vote
011	CCV4DQ	Vote
100	CCCV2ci	A → CPU
101	CCCV1ci	CPU → A
110	SCSR	C → A
111	CCCV0ci	B → A

c = 1, doubleword

i = 11, doubleword

IV. INTERSTAGE

The interstage is the heart of the fault tolerant computing network. It provides the link between the CPU and two of the external channels in the SIR network. The interstage contains five 32-bit shift registers, four timer/counters, and utilizes a 32-bit internal bus. The interstage interfaces with two external serial input data lines and their respective clock signals. It further provides two output serial data lines and a clock. The interstage interfaces with the mid-value voter and receives instruction decoding and control signals from four 2Kx8 PROM controllers. The hierarchial interstage schematic, including the voter, is shown in Figures 4-1, 4-2, and 4-3.

A network to switch between integer and floating point representations while performing midvalue votes was part of the original design specification but the concept could not be realized because the LSTTL gates available in the SCALD CAD system were too slow for a 10 MHz clock. Section VI, "Voter", has the details.

A. 32-BIT SHIFT REGISTERS

The five 32-bit shift registers (Figure 4-4a) are labeled as A, B, B', C, and C'. Four LS299 8-bit Universal Shift/Storage Registers with Tri-state Outputs are used to

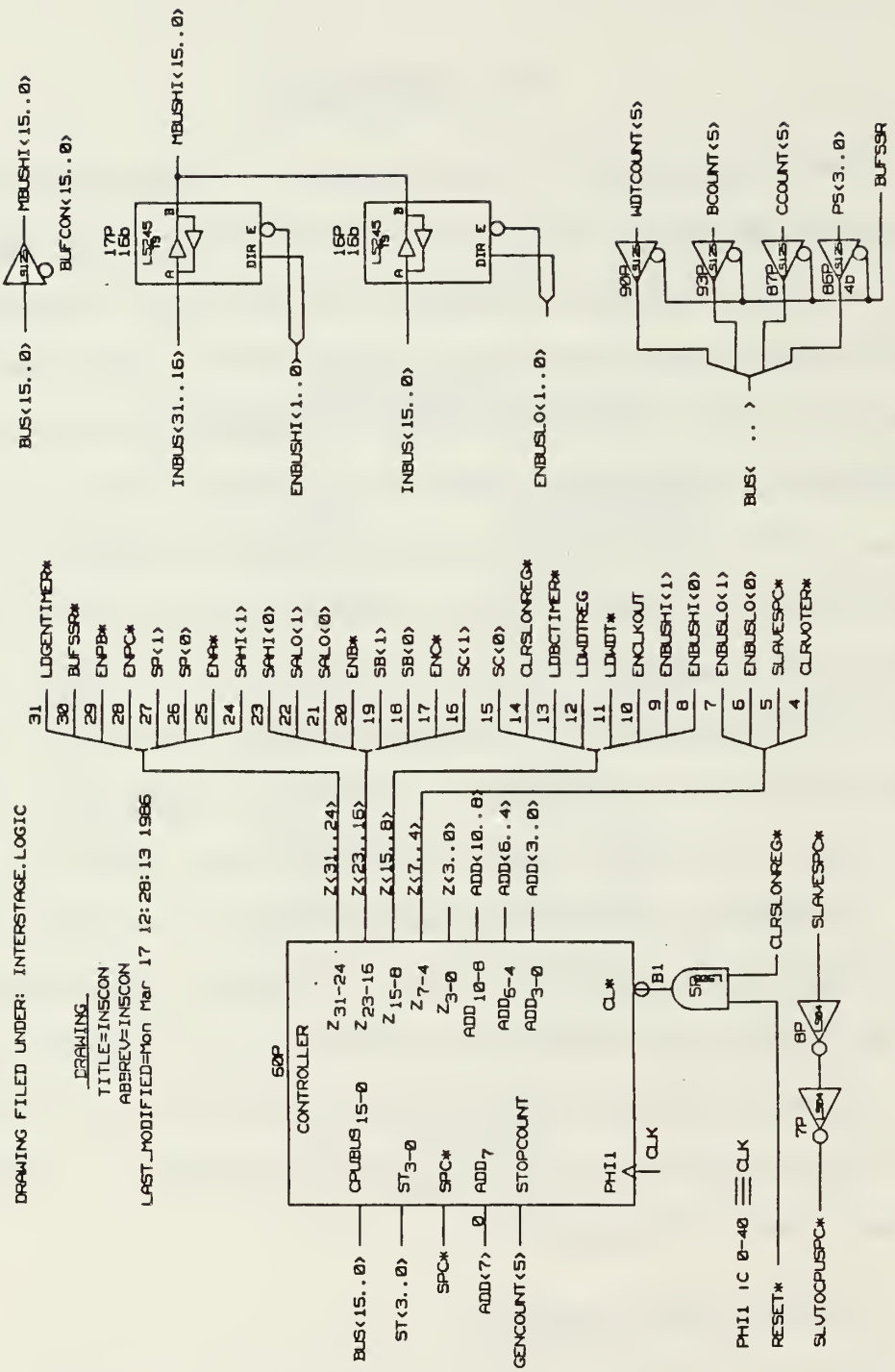


Figure 4-1: Interstage - Controller, SSR, and Bus Buffers

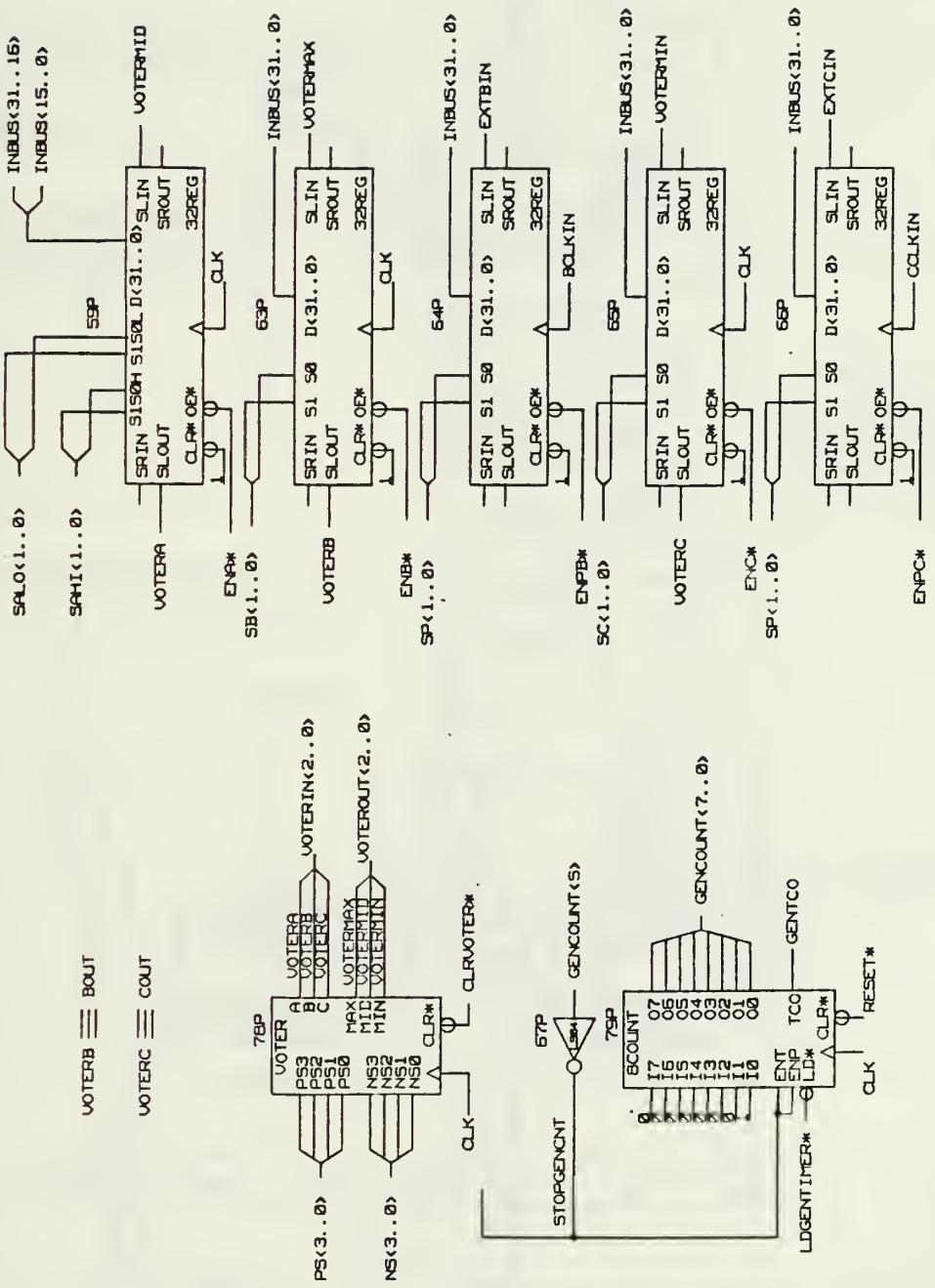


Figure 4-2: Interstage - Voter, GenCounter, and Registers

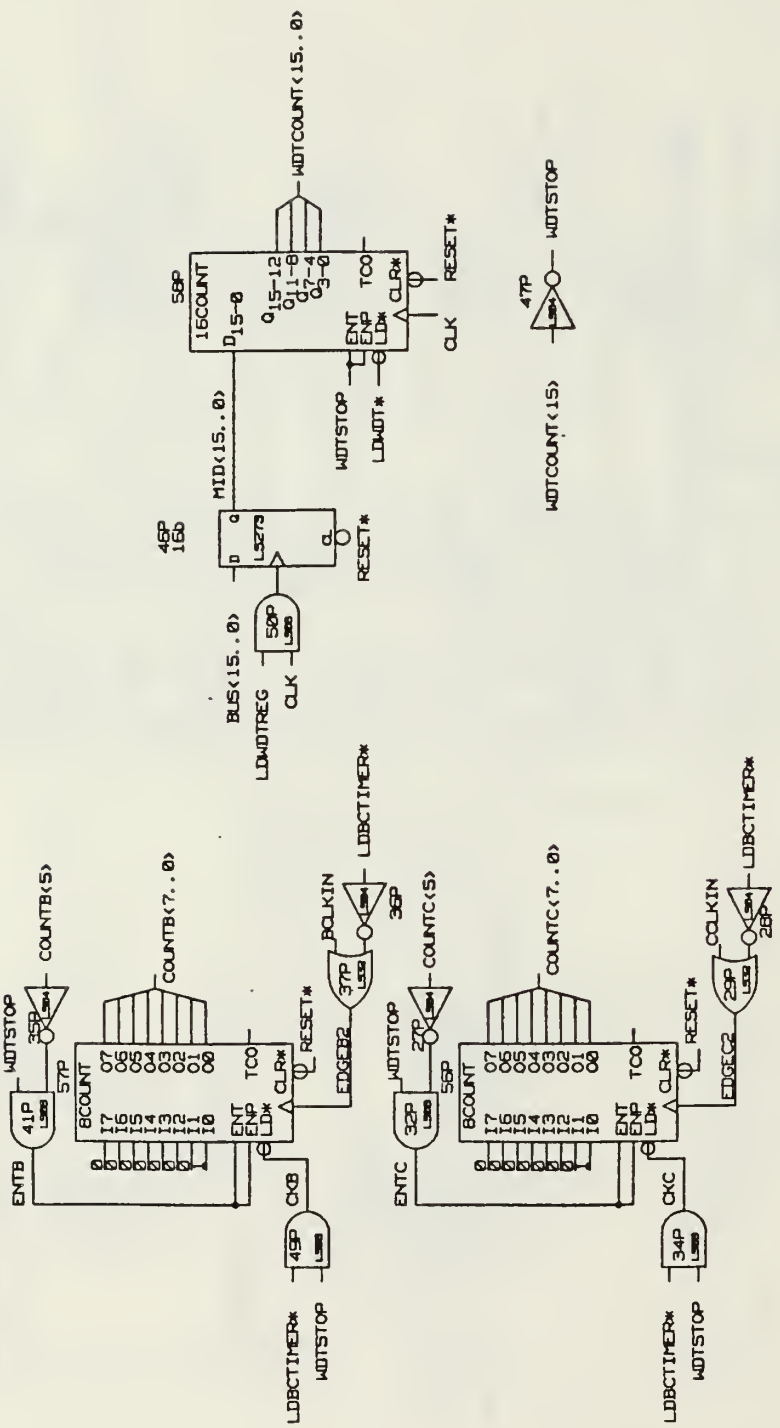


Figure 4-3: Interstage - Serial Input Counters

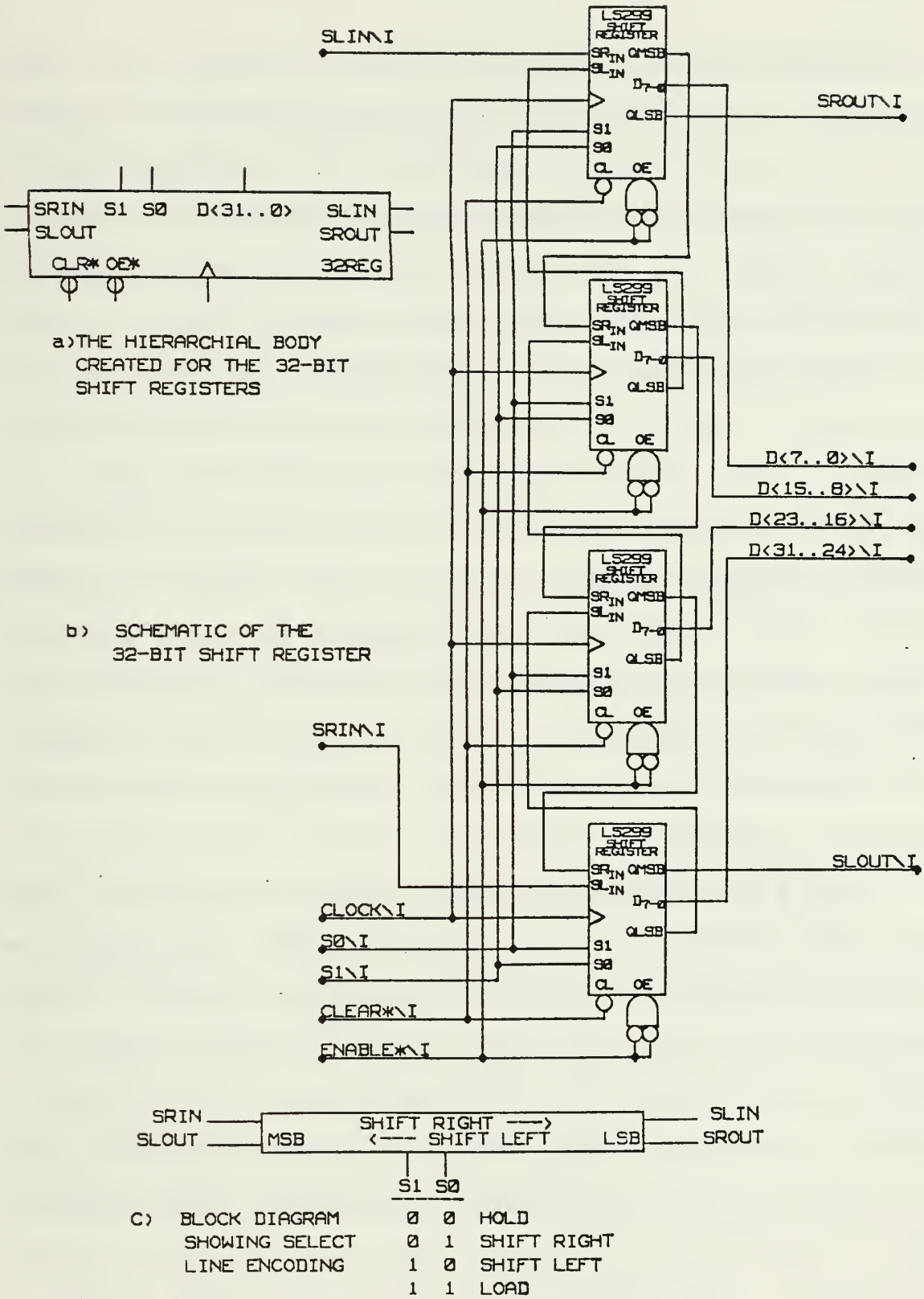


Figure 4-4: 32-bit Universal Shift/Storage Register

implement each 32-bit shift register (Figure 4-1b). Each register is connected to the interstage's 32-bit internal bus for parallel data transfers. The registers use two control lines: clear (CLR*) and output enable (OE*); and two select lines. The two select lines (S1 and SØ, Figure 4-4c) can select shift left, shift right, parallel load, or hold.

The A register can be loaded in four separate ways (3 parallel, 1 serial). The first is by copying a doubleword from the CPU. The A register is the only register that directly transfers data to or from the CPU. The second, a serial transfer, occurs during a mid-value vote (instruction Ø11). Data is simultaneously serially shifted out to the voter and the result of the vote is serially shifted back. The final two load operations are parallel: copy B register to A register (instruction 111) and copy C register to A register (instruction 11Ø).

The B and C registers have two major functions. They serially transfer data out to the external interstages and store operands and results of mid-value voting. Three methods are used to load the B and C registers. The first is parallel loading from the prime registers (B' →B and C' →C). This is the first step of the "VOTE" instruction. The second is by instruction ØØØ, which takes the contents of the A register and parallel loads it to both the B and C registers. The final method is a serial shift-left load

from the output of the mid-value voter (instruction Ø11). Storing the results of the vote is identical to the procedure used by the A register.

The B' and C' registers receive serial input data from external interstages (see Section III B, "Serial Data Input"). Once the data has been received, it is held until the VOTE command is received. The data is then parallel loaded to the B and C registers.

The mid-value voter works directly with the A, B, and C registers. The three operands stored in the registers are serially transferred bit by bit to the mid-value voter and compared. The voted values (mid, max, min) are directed to the A, B and C registers respectively and stored. Each "shift left out" and "shift left in" operation takes place during one clock cycle. The bit by bit voter sequence is very similar to the actions of a ring counter. In the voter's case, one bit from each of the three shift registers are compared and the results of the comparison are recirculated back to the same shift registers. At the end of 32 counts, the comparison is terminated. The mid value is sent to the A register since it will be either loaded into the CPU or parallel recirculated into the B and C registers for transmission to the external interstages.

B. SERIAL DATA INPUT

One of the main functions of the interstage is to receive data from external sources, manipulate, then subsequentially transfer the data out. Incoming serial data is loaded into a 32-bit shift register under the control of an 8-bit, modulo-32 counter (see Figure 4-5) and a 16-bit counter called the Watch Dog Timer (WDT) (see Figure 4-6).

The incoming serial data is received independently from two channels: "Channel B In" and "Channel C In" (see Figure 4-7). The independent data streams are loaded into the B' and C' registers while a modulo-32 counter for each channel monitors the transfers. Two counters are required since the two channels operate independently of each other. Each channel has a clock provided by the sending external interstage.

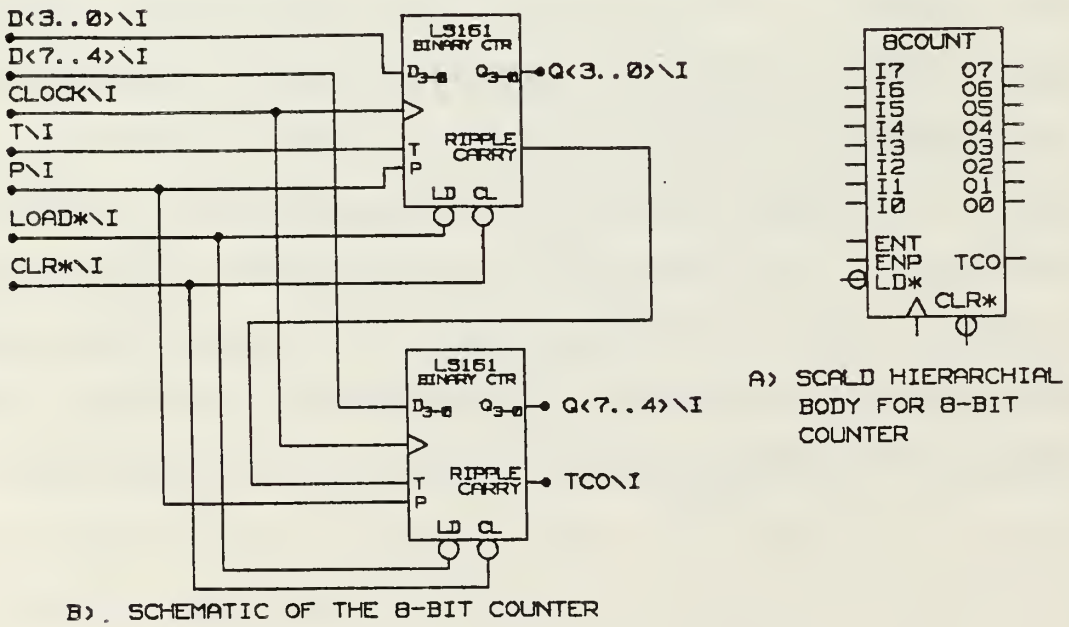
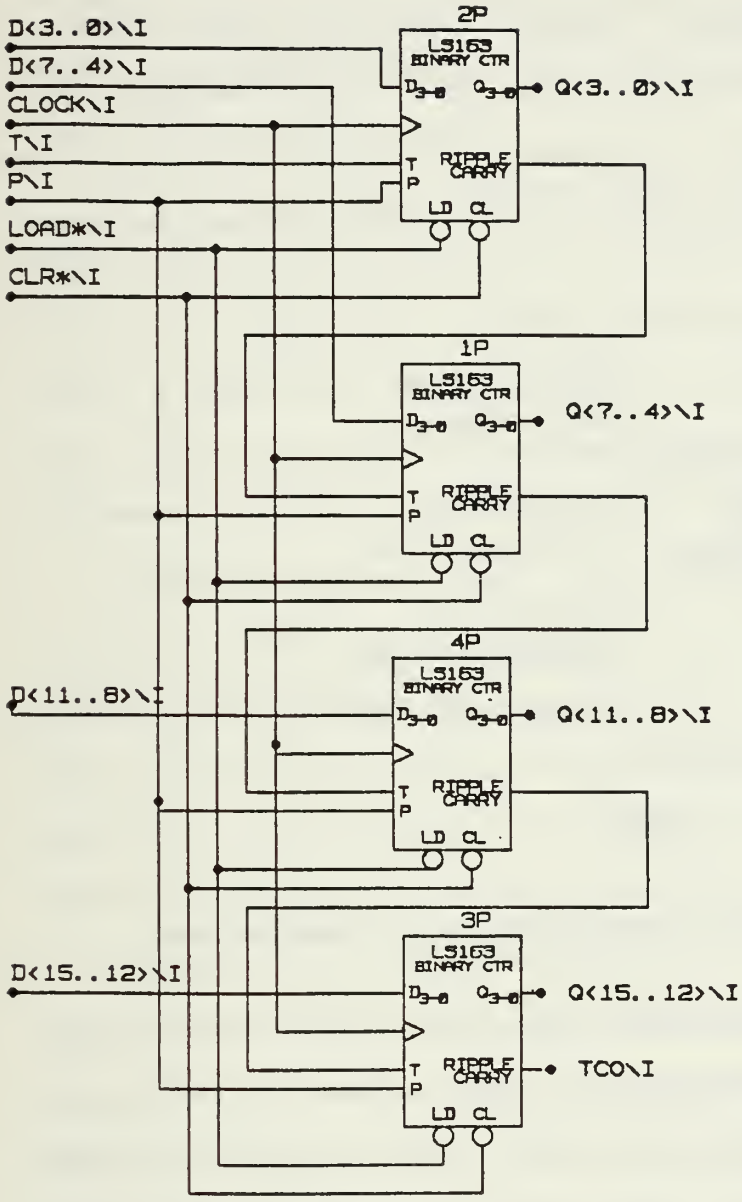
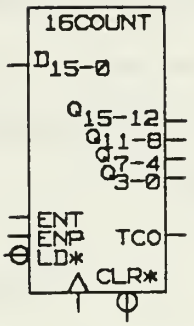


Figure 4-5: Eight Bit Counter



B) SCHEMATIC FOR THE 16-BIT COUNTER



A) HIERARCHIAL BODY FOR 16-BIT COUNTER

Figure 4-6: 16 Bit Watch Dog Timer

There are four control signals associated with each 8-bit mod-32 counter: CLR*, LD*, ENT and ENP. The CLR* line is permanently set high as there is no need to clear the counter. A preset value of "1" is loaded using the load command.

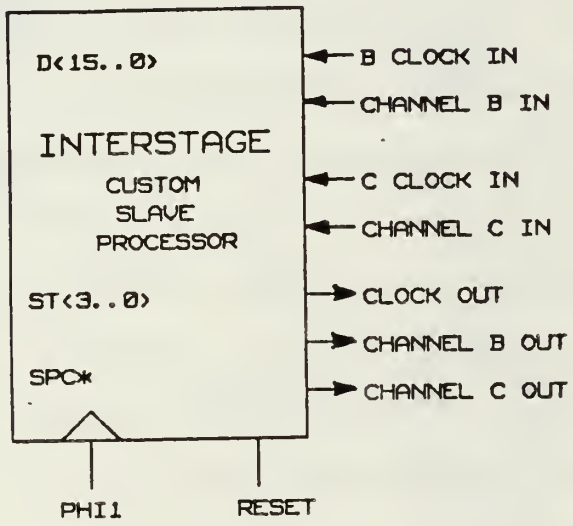


Figure 4-7: Serial Input/Output to Interstage

The LD* signal has a dual function. When LD* is low, the counter presets a "1" as the start value for the count. Holding LD* low provides the side benefit of placing the counter in a wait state. Even if the counter is clocked, the count does not advance.

When LD* is high (and ENT and ENP are high) the counter counts. Thirty two counts later, the COUNT<5> line (bit five of the counter) goes high and suspends the count. This line also acts as a flag to signal that 32 bits have been loaded into the register. The clock lines for the input

channels are held low until the external interstages are ready to transfer data. A transfer is initiated when the clock lines are enabled.

The ENT and ENP control lines are wired together. Both lines must be high in order to count. An inverter and an AND gate (Figure 4-3) disables the counter after 32 counts. This is important since an extra clock pulse or noise spike will cause the 32-bit register to serially load extra bits and store an inaccurate value into the register.

Serial loading the B' and C' registers requires the incoming clock line be held low until the external device is ready to transfer data. The first bit must be stable on the data line when the first rising clock edge is received. This stability is guaranteed since the other interstages are identical to the one in this thesis.

There is no synchronization between the two input channels. Conceivably, any combination of clock lines or channels could fail. The watch dog timer (WDT) provides a measure of control over the two incoming clocks (see Figure 4-3). The WDT starts counting immediately after system reset and continues counting until suspended (or reset at the end of a vote).

If one or both clocks fail to start in a predetermined amount of time, the clock failure is recognized as a failure by that channel. This failure is noted since the WDTSTOP

signal will go low and the counting operations of the BCOUNT and CCOUNT counters will be suspended. WDTCOUNT<15>, BCOUNT<5>, and CCOUNT<5> are sent to the interstage slave status register (SSR). The CPU checks the SSR at the end of a VOTE to determine the status of the data loaded into the B' and C' registers. Table 4-1 shows how the CPU would decode these bits.

The last two bits shown in Table 4-1 are the most important since the bits show whether a register is loaded or not. If both registers are loaded, it does not make a difference whether the window is open or closed. If one or both registers fail to load and the window is closed, this could indicate that an external circuit has failed. The CPU, by keeping track of which channel continually fails could eventually determine that something associated with the channel has failed. A vote should not begin until the window has closed. An interrupt or software timer should be used to ensure this.

If the input data is corrupted, this will be detected during the midvalue vote since the three 32-bit values will not be equal. The CPU, by reading and decoding the SSR, can determine which of the three channels contains the corrupted value.

The two interstage controller signal lines that control the serial data input are LDBCTIMER* (load the B and C registers) and LDWDT* (load the watchdog timer). When the

fault tolerant computer is initially turned on or reset, LDBCTIMER* is held low (wait state) and the modulo-32 counters are loaded with their start values of "1". By immediately enabling the counters, data can be loaded from the input channels independent of command or activity by the CPU or the interstage.

TABLE 4-1
SLAVE STATUS REGISTER DECODING

WDTCOUNT<15>	BCOUNT<5>	CCOUNT<5>	
Ø	Ø	Ø	Neither B' or C' loaded (Window open)
Ø	Ø	1	Only C' loaded (Window open)
Ø	1	Ø	Only B' loaded (Window open)
Ø	1	1	Both registers loaded (Window open)
1	Ø	Ø	Neither B' or C' loaded (Window closed)
1	Ø	1	Only C' loaded (Window closed)
1	1	Ø	Only B' loaded (Window closed)
1	1	1	Both registers loaded (Window closed)

LDWDT* is activated only by a CPU command. This instruction loads the watch dog register with its initial count. If no value is received from the CPU, the WDT uses the reset value of "ØØØØ" as an initial count. Upon completion of every serial-in transfer, the WDT is reset to

the inital count stored in the WDT register. Note that the loaded value is not the absolute count, but a relative count. When WDTCOUNT<15> goes high, it acts as a flag to indicate that the window to input serial data has closed. If a count of 50, 5 microseconds, is desired, 32,768 (2^{15} - 50) is loaded into the register. The WDT will continue using the same initial count value until reprogrammed by the CPU or the interstage is reset. It is imperative that the WDT register be loaded after every manual reset.

C. SERIAL DATA OUTPUT

Data is serially transferred out from the B and C registers (not to be confused with the B' and C' registers). The three control signals involved are the two bit select lines for each register and ENCLKOUT (enable clock out). The data bits are counted by a general timer, GENTIMER, which is identical to the 8-bit modulo-32 counters explained previously (see Figure 4-5).

Serial output transfer of data can be performed only under command of the CPU (instruction 001). The clock for the output channel is kept low by an AND gate and the control line ENCLKOUT. When the data is ready to be transferred out, ENCLKOUT goes high and enables the clock. The BOUT and COUT (serial data output) lines are always attached to the (shift left) outputs of the 32-bit registers.

D. INTERNAL 32 BIT BUS & BIDIRECTIONAL BUFFERS

The interstage utilizes a 32-bit internal bus to transfer information among the five 32-bit registers. This 32-bit internal bus interfaces with the CPU's 16-bit bus through 32 74LS245's, Octal Bus Transceivers with tri-state outputs (Figure 4-1). The bidirectional bus has two control lines, EN and DIR (enable and direction, Table 4-3).

During normal CPU operations, the 74LS245's are in the isolate condition buffering the interstage off the CPU's 16-bit external address/data bus. By dividing the 32 74LS245's into two sets of 16 each (16-bit select high and 16-bit select low), the transceivers act as a multiplexer converting the interstage's 32-bit bus format to the CPU's 16-bit external bus format.

TABLE 4-3
CONTROL SIGNALS FOR INTERSTAGE BI-DIRECTIONAL BUFFERS

EN	DIR	
Ø	Ø	A - B
Ø	1	B - A
1	X	ISOLATE

E. INTERSTAGE SLAVE STATUS REGISTER (SSR)

The SSR (see Figure 4-1) has seven data bits, each being the output of a register in the interstage. As such, the SSR is not an independent register. The control signal

BUFSSR (buffer for the slave status register) controls seven LS125 buffers. The buffers are enabled onto the high byte of the CPU's 16-bit external bus only when the interstage is passing status to the CPU during the VOTE command.

The four lower bits of the SSR are the present state, PS<3..Ø>, output lines from the voter (see Figure 4-6). The CPU can decode these bits and determine which of the 13 states the voter was in at the completion of a VOTE.

The next two bits are the BCOUNT<5> and CCONT<5> lines from the BCOUNT and CCOUNT modulo-32 counters. The final bit is the COUNT<15> line from the WDT. By decoding the three counter bits, the CPU can determine if one, both, or neither of the external channels were loaded correctly and within the WDT's window.

V. INTERSTAGE CONTROLLER

The interstage controller is the link between the CPU and the interstage. The controller receives instructions from the CPU and produces the control signals needed by the interstage to perform the required operation.

The controller consists of decoding logic, instruction registers and a 2K x 32 PROM. Appendix C contains the PROM memory contents. The actual chip count is provided in Table 5-1. The scald hierarchial body and the CPU interface of the controller is shown in Figure 5-1. The controller remains in a WAIT state until given a command by the CPU.

TABLE 5-1
INTERSTAGE CONTROLLER HARDWARE REQUIREMENTS

Gates	Chips	Chip	Description
9	2	LS175	Quad D flipflops
2	1	LS74	Dual D flipflops
4	4	27S291	2K x 8 PROM
9	2	LSØ4	Hex Inverters
2	1	LS11	Quad 3-Input AND
1	1	LS3Ø	8-Input NAND
2	1	LS2Ø	Dual 4-Input NAND
1	1	LSØ2	Dual 2-Input NOR

When the system is initially turned on or reset, the following actions occur:

(1) clears the instruction register

(2) clears the FSM state registers

(3) clears the SLAVEON and READY registers

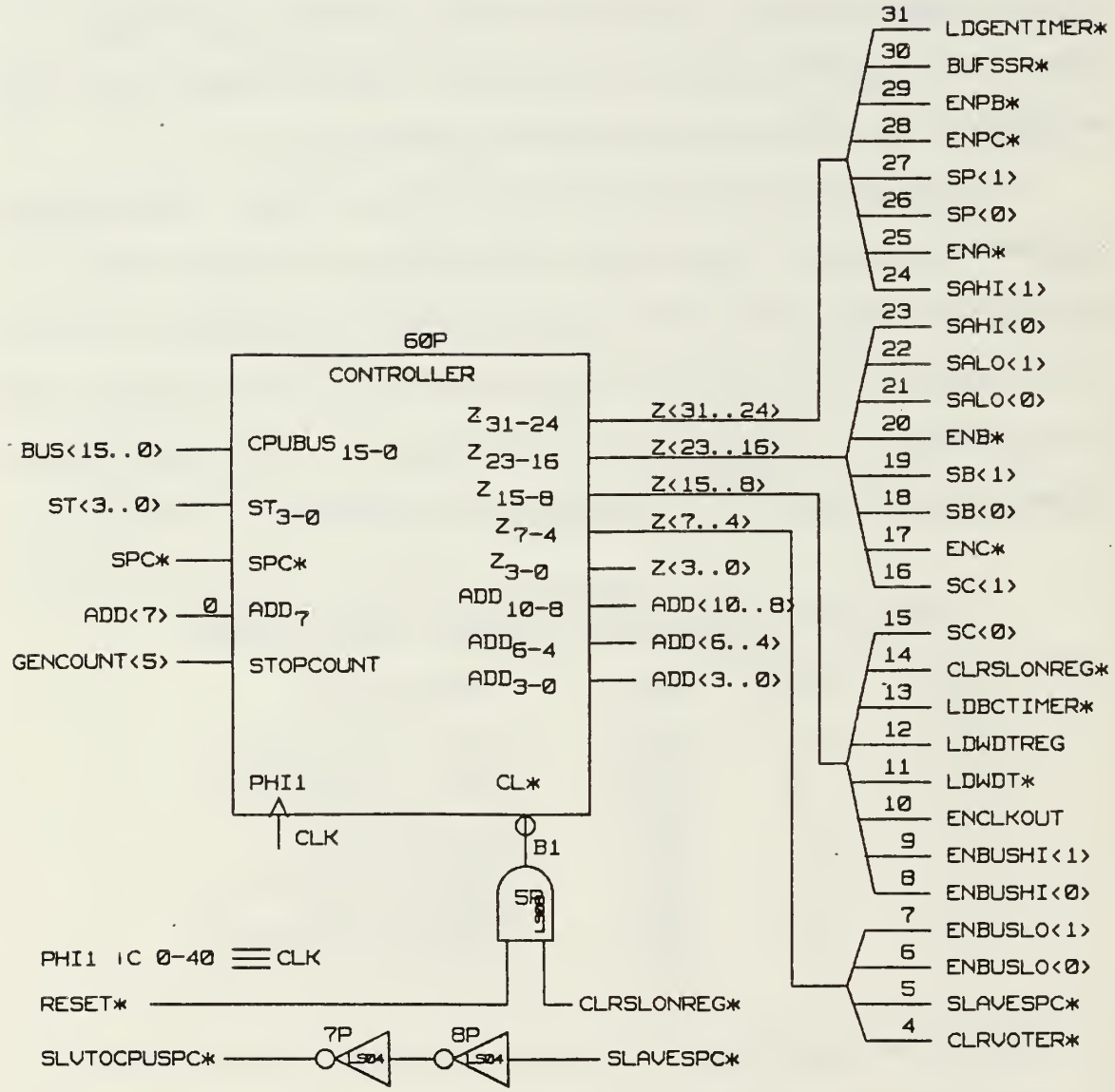


Figure 5-1: Interstage Controller and CPU Interface

The controller then continuously cycles through a wait state, AD<1Ø..Ø> = ØØØh (h=hex). The wait state performs the following:

(1) buffers the interstage off the CPU's external bus

(2) loads the three modulo-32 timers

(3) allows the B' and C' registers to shift left in

(4) enables the watchdog timer

The wait state sets the B' and C' timers so that serial data can be loaded from external interstages independent of CPU control. It is important to remember that the Watchdog timer register must be loaded (instruction Ø1Ø) after each reset for the relative count to be meaningful.

The controller interfaces with the CPU by monitoring the 4-bit status lines ST<3..Ø>, the 16-bit address/data bus, and the Slave Processor Control line (SPC*). The controller uses a clock, PHI1, synchronized with the PHI1 used by the CPU. The RESET signal should be the same signal used to reset the CPU and the entire system.

A. DECODER AND INSTRUCTION REGISTER

The controller remains in the wait state until activated by the CPU. The CPU selects the interstage by simultaneously transmitting the broadcast ID (1111) over ST<3..Ø> and the appropriate ID byte over AD<7..Ø>. The

controller decodes these 12 bits and the interstage is selected if the following bits are received:

ST<3..Ø>	A/D<7..Ø>
1 1 1 1	n n n 1 Ø 1 1 Ø

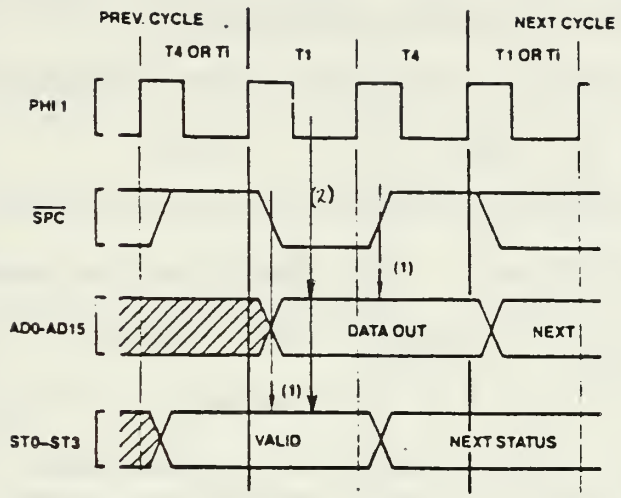
where nnn describes a particular operation word format. Section III, "Interstage Instruction Set", explains why format nnn = ØØ1 was chosen . Therefore, the following 12-bit sequence uniquely selects the interstage:

ST<3..Ø>	A/D<7..Ø>
1 1 1 1	Ø Ø 1 1 Ø 1 1 Ø

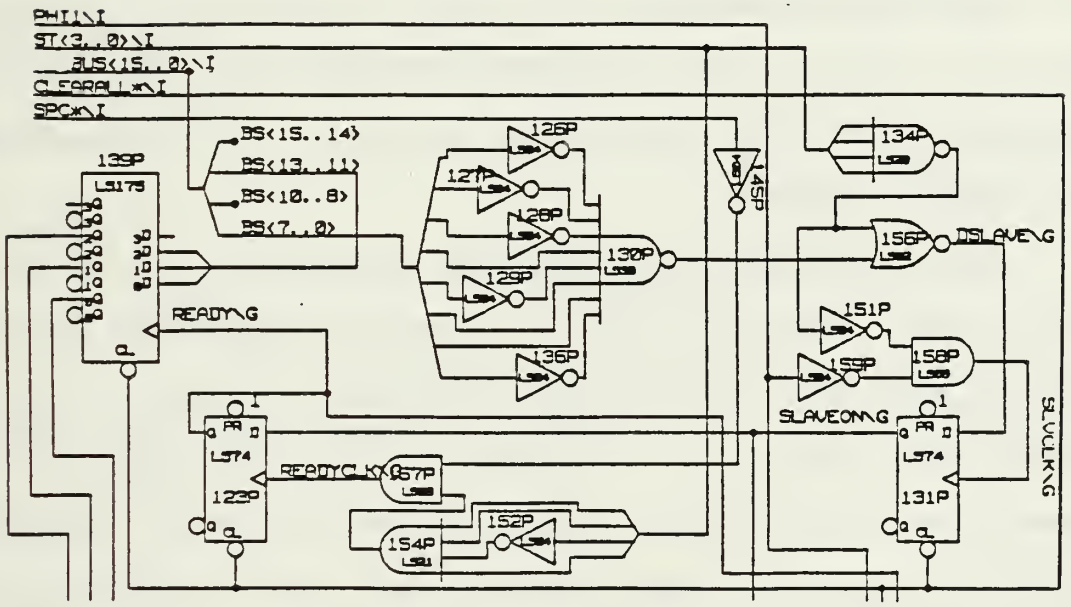
The decoding logic used to signal selection of the interstage is shown in Figure 5-2. The SLAVEON register (Figure 5-2, 131p) is set high only if the interstage is selected by the CPU. The READY register (Figure 5-2,123p) is set only if the SLAVEON register is set and when ST<3..Ø> is equal to "11Ø1".

The 12-bit selection sequence is decoded by the gates shown in Figure 5-2 (130p, 134p and 156p). The output, DSLAVE, is high only when the signal to select the interstage is transmitted by the CPU. DSLAVE is the input signal for the SLAVEON register.

The clock signal for the SLAVEON register, SLVCLK, is the output of a 2-input AND gate (Figure 5-2, 158p). The first input acts as an enable that is high only if the



a) CPU Write to Slave Processor



b) Controller Hardware

Figure 5-2: Selection Decoder for Interstage Controller

output of the 4-input NAND (Figure 5-2, 134p) is high. Otherwise, SLVCLK is low, SLAVEON is not set and the interstage is not selected. The other input is the clock signal inverted.

Figure 5-2 (a) shows the timing requirments to sample ST<3..0> and AD<15..0>. By sampling at point (2) and holding for the remainder of the negative clock cycle, DSLAVE will be high for approximately 50 nanoseconds. SLVCLK goes high approximately 10 nanoseconds later due to the propagation delay of the AND gate 158p and provides a positive rising edge for the SLAVEON register. At this point, the SLAVEON register is set indicating selection of the interstage by the CPU for slave instructions.

The timing diagram in Figure 5-3 shows the selection process. During the T1/T4 cycle from 400 - 600 ns, the controller correctly decodes the data from the CPU and sets the SLAVEON register.

The CPU sends the operation word on the next T1/T4 cycle (600 - 800 ns). ST<3..0> transmits 1101 and the low byte of the operation word is swapped and sent over the address/data bus (bits 15-8 appear on A/D<15-8> and bits 7-0 appear on A/D<23-16>). This is an important sequence since the operation word contains the instruction and will be clocked into the 3-bit instruction register.

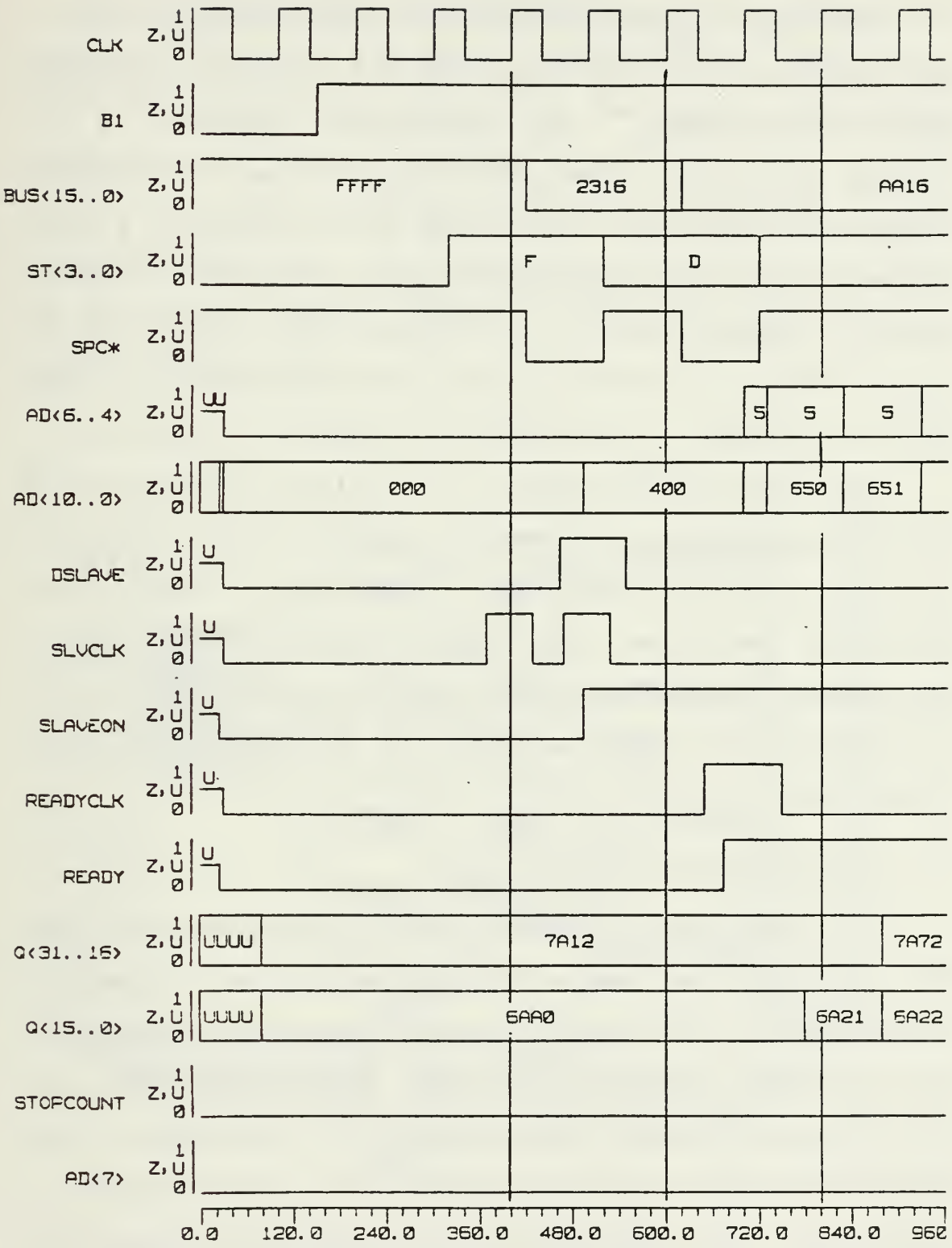


Figure 5-3: Timing Diagram for Interstage Selection

The READY register (Figure 5-2, 123p), initially set to zero, provides the clock signal for the instruction register (Figure 5-2, 139p). The instruction register is not clocked unless the SLAVEON register is set. The READY register changes from low to high only once during a slave processor sequence providing a postitive edge triggered pulse to clock in the 3-bit instruction. Figure 5-2 (a) shows that the instruction should be clocked during T4 and this is verified in the simulation (AD<6..4>, Figure 5-3).

The output of the instruction register provides three bits of the address required for the Finite State Machine portion of the the sequential controller. None of the registers described thus far will change state until completion of the slave cycle. At that time, they will be cleared and will remain so until the interstage is selected by the CPU.

B. FSM PORTION OF THE CONTROLLER

The interstage can respond to eight different commands. Once the controller has decoded an instruction, it operates independently of the CPU while executing the given instruction. Once the CPU has prefetched the next instruction, further CPU operation is suspended. Upon completion, the controller drives SPC* low (during the T1 cycle only) and transfers the slave processor status word and operands back to the CPU as dictated by the instruction.

The finite state machine portion of the controller consists of four 2K x 8 27S291 PROM's and six 74LS175 D-flipflops (Figure 5-4). Paralleling the eleven address bits across the four PROM's provides 32 control lines. Of the 32-bits of output, four are used as the next state inputs for the D-flipflops. The other 28 bits are available for use as active control lines for the interstage and are listed in Table 5-2.

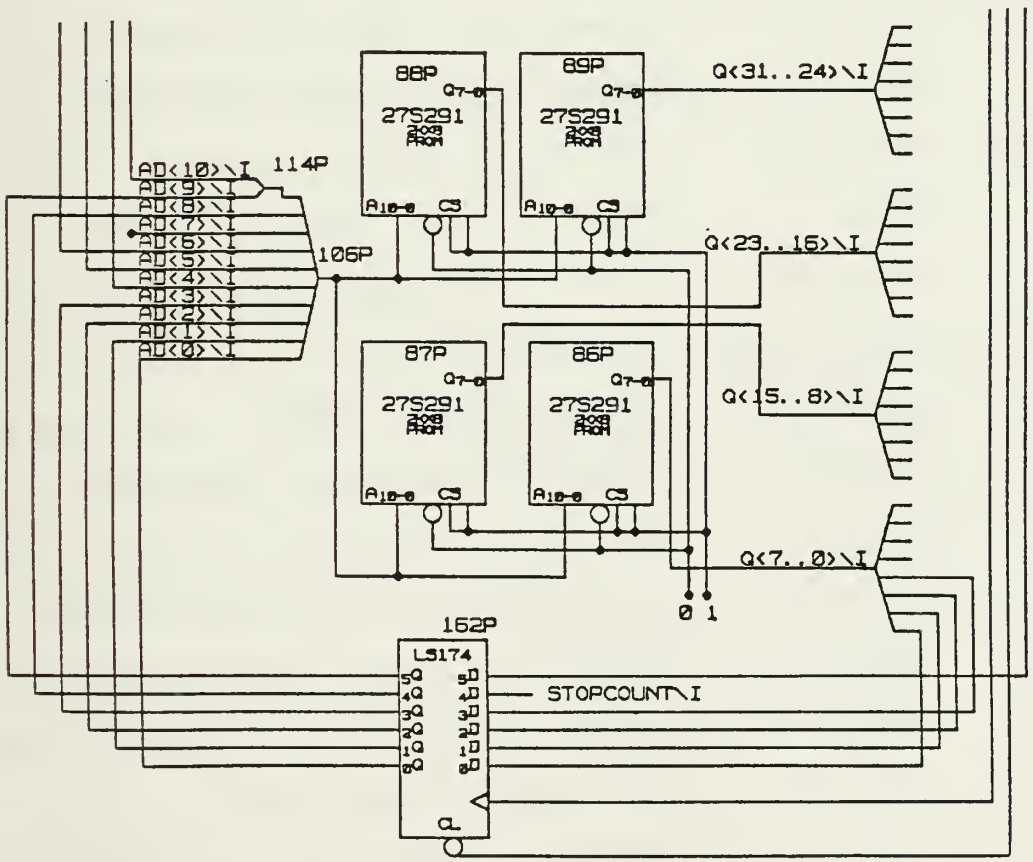


Figure 5-4: FSM Portion of the Controller

TABLE 5-2
32 CONTROL SIGNALS FOR THE INTERSTAGE

PROM BIT	NAME	DESCRIPTION
Q<31>	LDGENTIMER*	Load the General Timer
Q<30>	BUFSSR*	Enable the SSR buffers
Q<29>	ENPB*	Enable 32 bit output of B' reg
Q<28>	ENPC*	Enable 32 bit output of C' reg
Q<27..26>	SP<1..0>	Select lines for B' and C' reg
Q<25>	ENA*	Enable 32 bit output of A reg
Q<24..23>	SAHI<1..0>	Select for bits 31-16 of A reg
Q<22..21>	SALO<1..0>	Select for bits 15-0 of A reg
Q<20>	ENB*	Enable 32 bit output of B reg
Q<19..18>	SB<1..0>	Select for B reg
Q<17>	ENC*	Enable 32 bit output of C reg
Q<16..15>	SC<1..0>	Select for C reg
Q<14>	CLRSLONREG*	Clear SLAVEON reg
Q<13>	LDBCTIMER*	Load B & C reg timers
Q<12>	LDWDTREG	Load WDT reg with initial count
Q<11>	LDWDT*	Load WDT with initial count
Q<10>	ENCLKOUT	Enable output clock (serial out)
Q<9..8>	ENBUSHI<1..0>	Select for bits 31-16, LS 245
Q<7..6>	ENBUSLO<1..0>	Select for bits 15-0, LS 245
Q<5>	SLAVESPC*	SPC* low (signal to CPU)
Q<4>	CLRVOTER*	Clear the voter state registers
Q<3..0>		Inputs to State registers

Execution of each of the eight instructions requires sequential manipulation of the 28 control lines. The PROM is programmed to provide the control line values. The eleven address bits are described in Table 5-3.

TABLE 5-3
ADDRESS BITS FOR PROM CONTROLLER

AD<1Ø>	Output of SLAVEON register (high means begin)
AD<9>	Output of READY register
AD<8>	Stopcount (flag from mod-32 GENTIMER counter)
AD<7>	Not used (Set to zero)
AD<6..4>	Output of 3-bit instruction register
AD<3..Ø>	Present state for FSM

C. INITIAL STARTUP/RESET

A RESET, regardless of how initiated, will reset all the registers (with the exception of the WDT register) and load the timers with their initial value. The B' and C' shift register's select lines will be set to shift left. All other components are either disabled or placed in a hold state. These actions are accomplished during the wait state (Address ØØØh).

The controller will remain in the wait state until the interstage is selected by the CPU. For example, assume instruction "1Ø1" is sent to the interstage. When selected, AD<1Ø> goes high and the address changes to 4ØØh. The PROM

outputs remain the same. The READY register is set approximately 100 nanoseconds later and AD<9> goes high. Thirty nanoseconds later, the 3-bit instruction is clocked into the instruction register and the starting address for the instruction, 650h, is presented to the PROM.

Note that AD<9> goes high upon the rising edge of PHI1. Since READY is the clock for the instruction register, the instruction bits change approximately 25-35 ns after READY is set. This creates a race condition where AD<1..0> transitions from 000h - 400h - 600h - 650h. The race state is 600h. A race state appears in all eight instructions and is accounted for in the PROM with extra transition states. An alternate method to eliminate the race would be to use more registers to ensure that the address bits to the PROM all changed simultaneously.

The FSM cycles through its programmed states and produces the required control signals. When the instruction is complete, Q<5> (SLAVESPC*) and/or Q<14> (CLRSLONREG*) are pulsed low. When Q<5> goes to zero SPC* is driven low through a buffer (see Figure 5-5). Driving SPC* low signals the CPU that the interstage has completed the instruction. Q<5> and Q<14> pulse low simultaneously except after the VOTE command because the VOTE command passes operands to the CPU. Driving Q<14> low clears all registers in the controller and places the controller into the wait state.

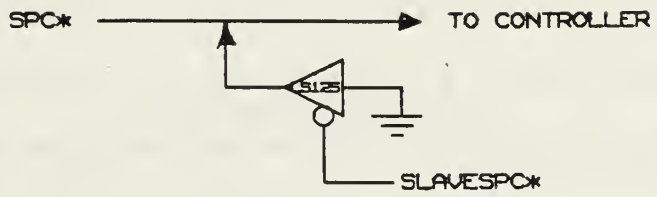


Figure 5-5: Network to Signal CPU that Instruction has been Completed

D. LOAD A REG →B,C REG (INSTRUCTION ØØØ)

Loading the contents of the A register into the B and C register takes only three clock cycles. The 74LS245 buffers isolate the interstage from the CPU since no information is transferred. The interstage ignores the doubleword the CPU is writing. ENA* is set low and the select lines for the B and C registers are set to "11", parallel load. The instruction is then finished. SPC* is pulsed low, and the interstage returns to a wait state.

Figure 5-8 is the timing diagram from the computer simulation. Instruction execution lasts from approximately 2ØØØ ns to 23ØØ ns. At 122Ø ns, the A register is enabled and its contents, Ø1234567h, can be read from INBUS<31..Ø>. This operation is independent of the instruction and is performed only to show the contents of the previously loaded A register. After execution of the instruction, the B register is artificially enabled (244Ø ns) and the C register is enabled (254Ø ns) to show their contents. Both

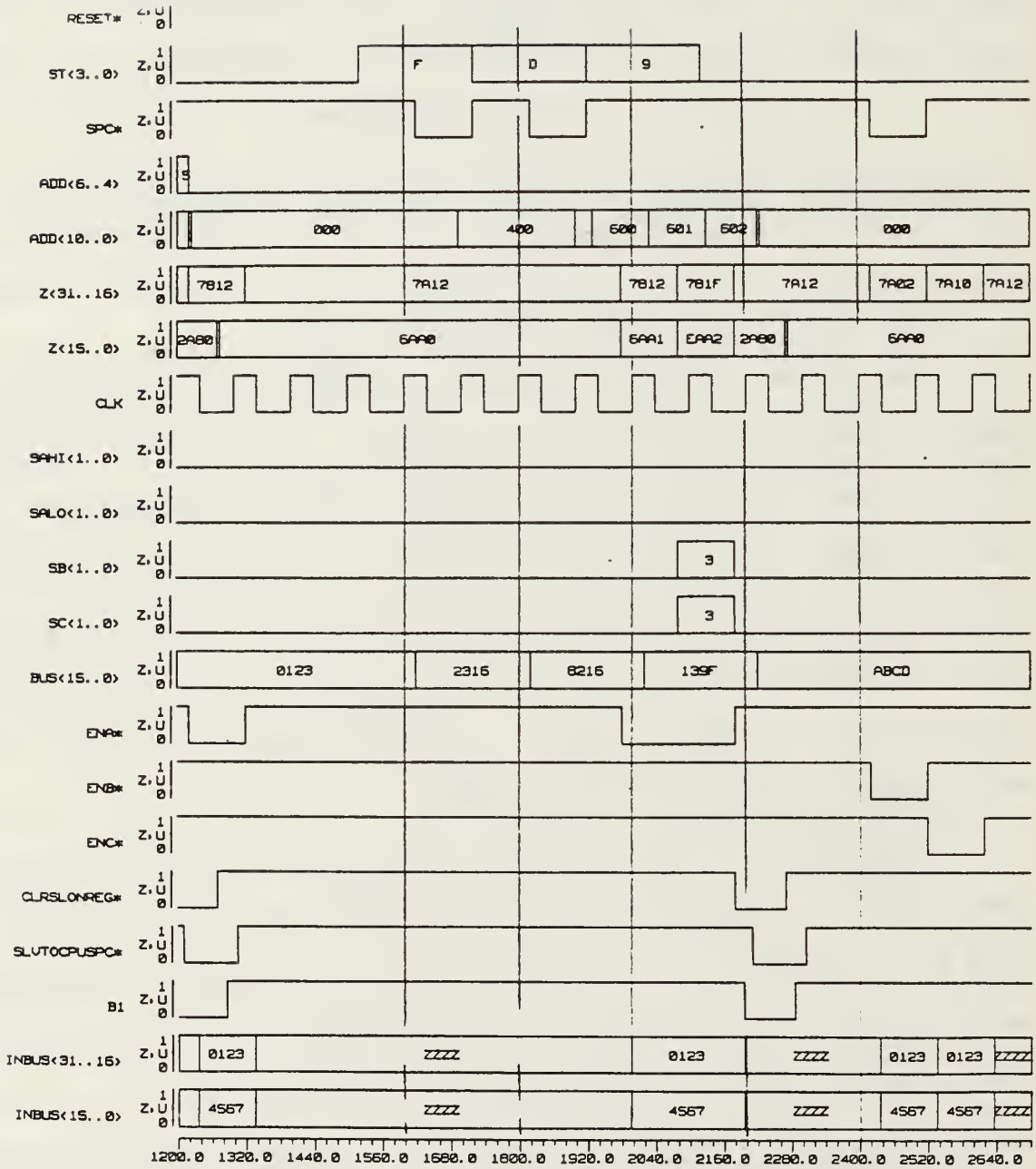


Figure 5-6: Computer Simulation of Instruction 000

registers contain Ø1234567h and therefore validate the instruction. Script files are used by the SCALD system to insert a series of logical values into the circuit under test and generate outputs over a specified period of time. The script file used for the simulation, "inØsim.dat", is in Appendix D.

After the instruction is completed, the controller returns to the wait state ØØØh at 222Ø ns. However, there is another race condition when transitioning from the last command address to the wait state. Since the SLAVEON and instruction register clears a few nanoseconds before the state register, a race condition is created. This race condition occurs at the end of each instruction for all eight instructions and extra transition states in the PROM account for it. As stated before, an alternate method to eliminate the race is to add registers.

E. SERIALLY TRANSFER OUT B,C REG (INSTRUCTION ØØ1)

This instruction was to have enabled the output clock (Q<1Ø>, ENCLKOUT) and serially transferred the contents of the B and C registers to the external interstages. While writing and simulating the VOTE instruction (Ø11) and the controller commands for autonomous serial loading of the B' and C' registers, timing problems developed with the general modulo-32 counter (GENTIMER). While the VOTE and serial

loading instructions did work, the addition of a third function to the general timer disrupted the timing.

The addition of multiple enables to allow the GENTIMER to perform three functions could not be accomplished in a simple manner. The only way to make this instruction work is to add two more mod-32 counters. This expansion also requires additional control lines. The hardware additions include four 74LS161 counters, another PROM and several gates for an enable/disable network.

At this point it became obvious that the interstage as originally conceived was becoming too large and therefore was affecting it's reliability. The hardware expansion required to make the command work was not implemented. Therefore, there was no need to write and simulate this simple controller sequence (enable the output clock and shift out). The discovery of the timing problem and its effect on the interstage is discussed further in Section VII, "Summary and Conclusions".

F. LOAD WDT / FP VS. INTEGER VOTE (INSTRUCTION 010)

Loading a start count for the watchdog timer should be the first command executed after any reset. Otherwise the watchdog counter starts from zero. Instruction 010h transfers a doubleword to the interstage. The lower word contains the value to be loaded into the watchdog timer register. The lower bit of the upper word contains a one

bit code for integer vote or floating point vote. As explained in Section VI, "Voter", the floating point vote could not be realized using the LSTTL components available in the SCALD CAD machine's library. Therefore, no use is made of the higher word transferred from the CPU.

The simulation for loading the WDT register is shown in Figure 5-7. The instruction has eight states. From 186Ø ns to 2ØØØ ns, the lower word of the interstage's 32-bit internal bus is enabled and the load watchdog timer register signal, LDWDTREG, pulses. At 2Ø8Ø ns, the watchdog timer is loaded with the new value when LDWDT* is pulsed low. From 215Ø ns forward, the watchdog timer will start counting at the loaded value (7FCØh in this simulation). Notice that the signal WDTCOUNT<15..Ø> starts counting from zero, then jumps to 7FCØh and continues counting from there. The last two states are not shown, but that is where Q<5> and Q<14> pulse low and reset the controller. The script file is in Appendix D.

G. VOTE (INSTRUCTION Ø11)

The vote instruction is the most complicated of the eight. The controller first loads the contents of the B' register to the B register and the contents of the C' register to the C register. The A register should already contain a value (from instruction 1Ø1). Next, under the control of the general timer, the 32-bit serial vote of the

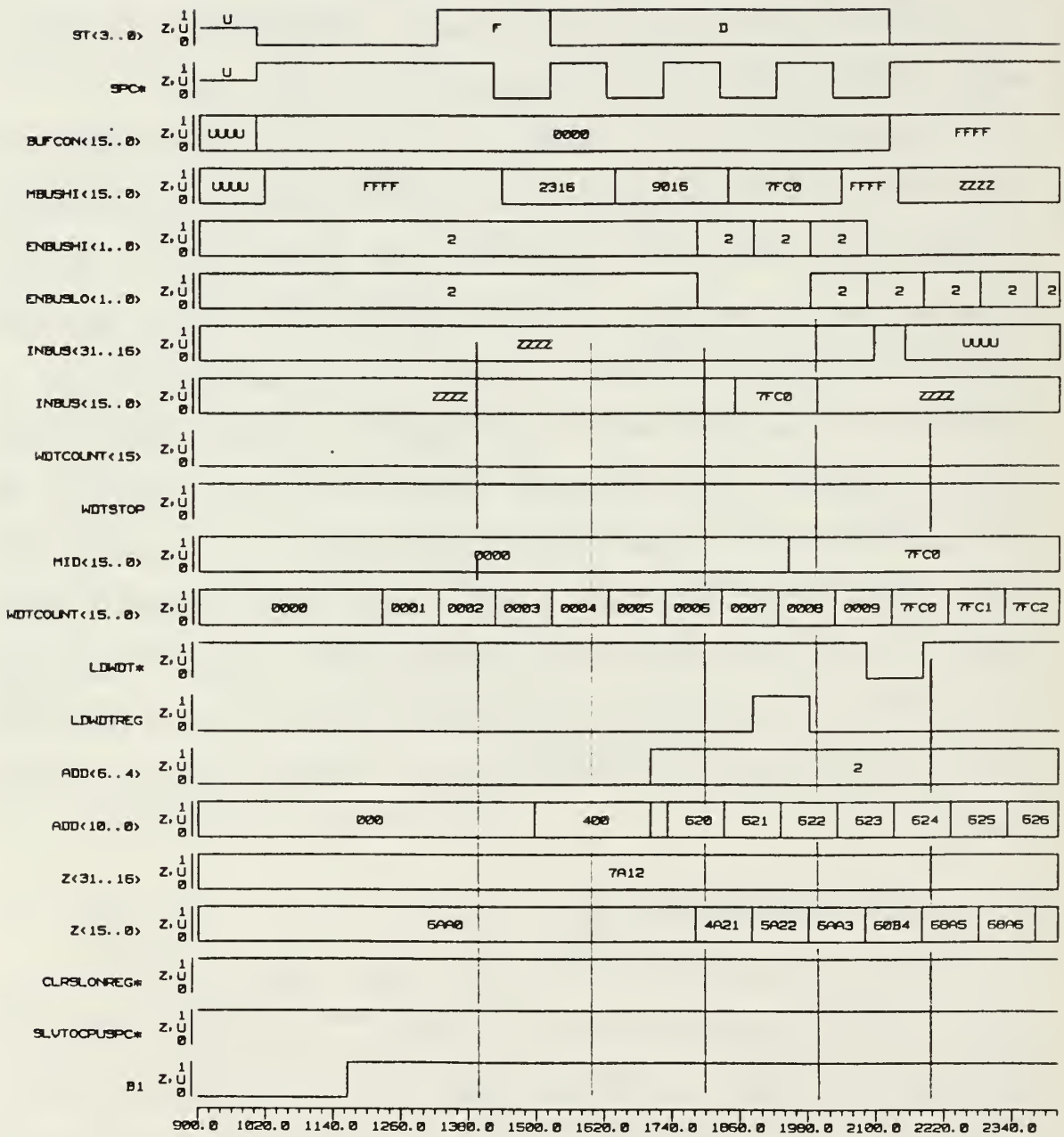


Figure 5-7: Computer Simulation for Instruction 010

three registers begins. After completion of the vote, the mid value is in the A register, max in the B register, and the min in the C register. The contents of the A register and the slave status register (SSR) is then read by the CPU.

Figures 5-8, 5-9, 5-10, and 5-11 are the timing diagrams for Instruction 011. Figures 5-8 and 5-9 both start at 6100 ns and end at approximately 8340 ns. Figures 5-10 and 5-11 start at approximately 10100 ns and end at 11980 ns. The four figures must be viewed as a whole with Figure 5-8 above Figure 5-9, Figure 5-10 above Figure 5-11, with both pairs side by side.

For the computer simulation, the A register was previously loaded with the value 10AA5A5Ah. The B' and C' registers were loaded from a simulated external interstage. Their stored values are 62329697h and 6621696Bh respectively as shown on the internal bus, INBUS<31..0> (Figure 5-8), from 6100 - 6400 ns. The VOTE instruction is decoded at 7150 ns. INBUS<31..0> shows the B' and C' registers enabled on the bus starting at 7300 ns. The actual voting begins at 7500 ns.

An LSTTL implementation of the voter does not work with a 10 MHz clock. However, as pointed out in Section VI, "Voter", STTL and FAST implementations can work at 10 MHz. Figure 5-9, starting at 7500 ns, shows the problem. At 7500 ns, the VOTERIN<2..0> values are equal and the present state is correct at 0000h. However, at 7600 ns, the input is 011.

The A value is min and the next state should be 1100 (See Figure 6-5 and Table 6-2). The simulation shows the state clocked to the voter registers to be 0001. This is incorrect. The VOTEROUT<2..0> output, which should be 110, bounces from 000 to 111 to 000, all incorrect. The combinational logic in the voter does not have enough time to stabilize the outputs before the voter state register is clocked. If the clock for the system was expanded to 5 MHz, the VOTE instruction would work. This particular timing problem is thoroughly discussed in Section VI, "Voter". The vote problem is a function of the voter itself and the clock speed of the system and not with the interstage or the interstage controller.

Even though the output of the voter is "garbage", the controller continues. The controller remains in the 633h address during the 32 counts of the counter. At 10580 ns, the general counter reaches 20h. Thirty two bits have been circulated through the voter. GENCOUNT<5> goes high, Q<5> (the SPC* line) goes low, and the controller jumps to address 733h from 633h. STOPCOUNT in Figure 5-4, which is the same as GENCONT<5> in Figure 5-2 goes high. The results of the vote, the middle value, is transferred to the CPU (using the same sequence as instruction 100). The contents of the SSR is written to the CPU, Q<14> (CLRSLONREG*} is pulsed low, and the interstage resumes the wait sequence.

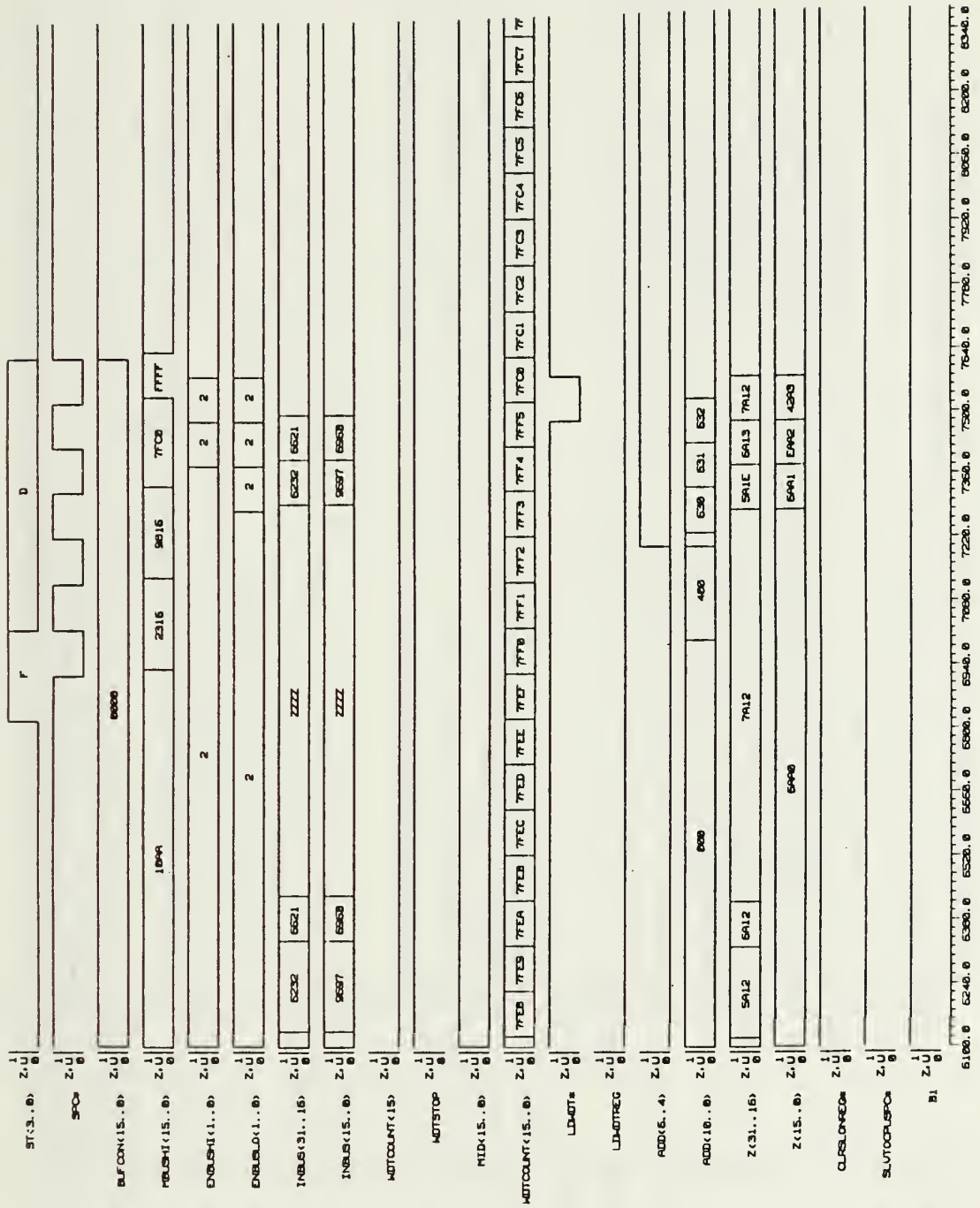


Figure 5-8: Computer Simulation for Instruction 011

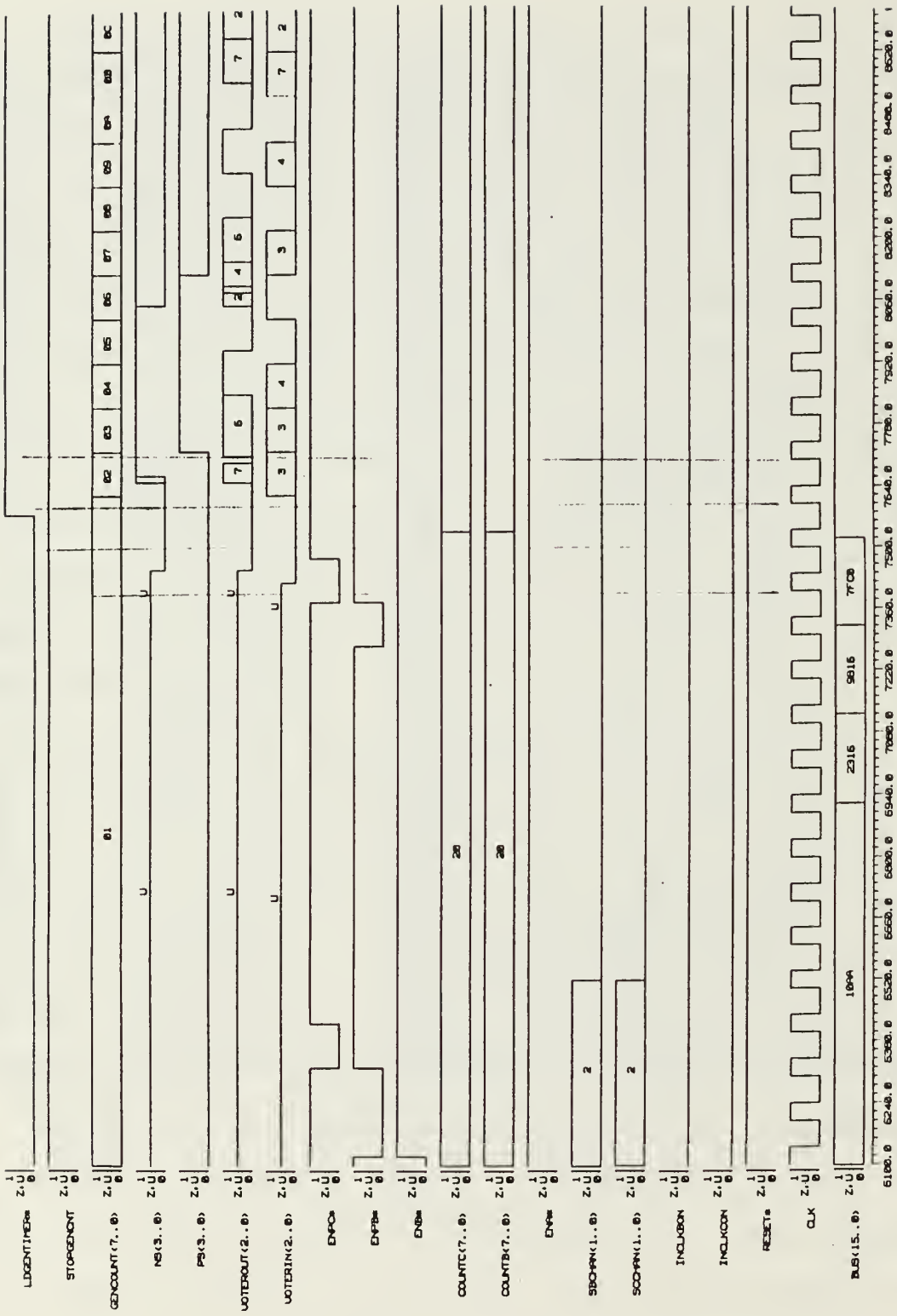


Figure 5-9: Computer Simulation for Instruction 011 (Cont)

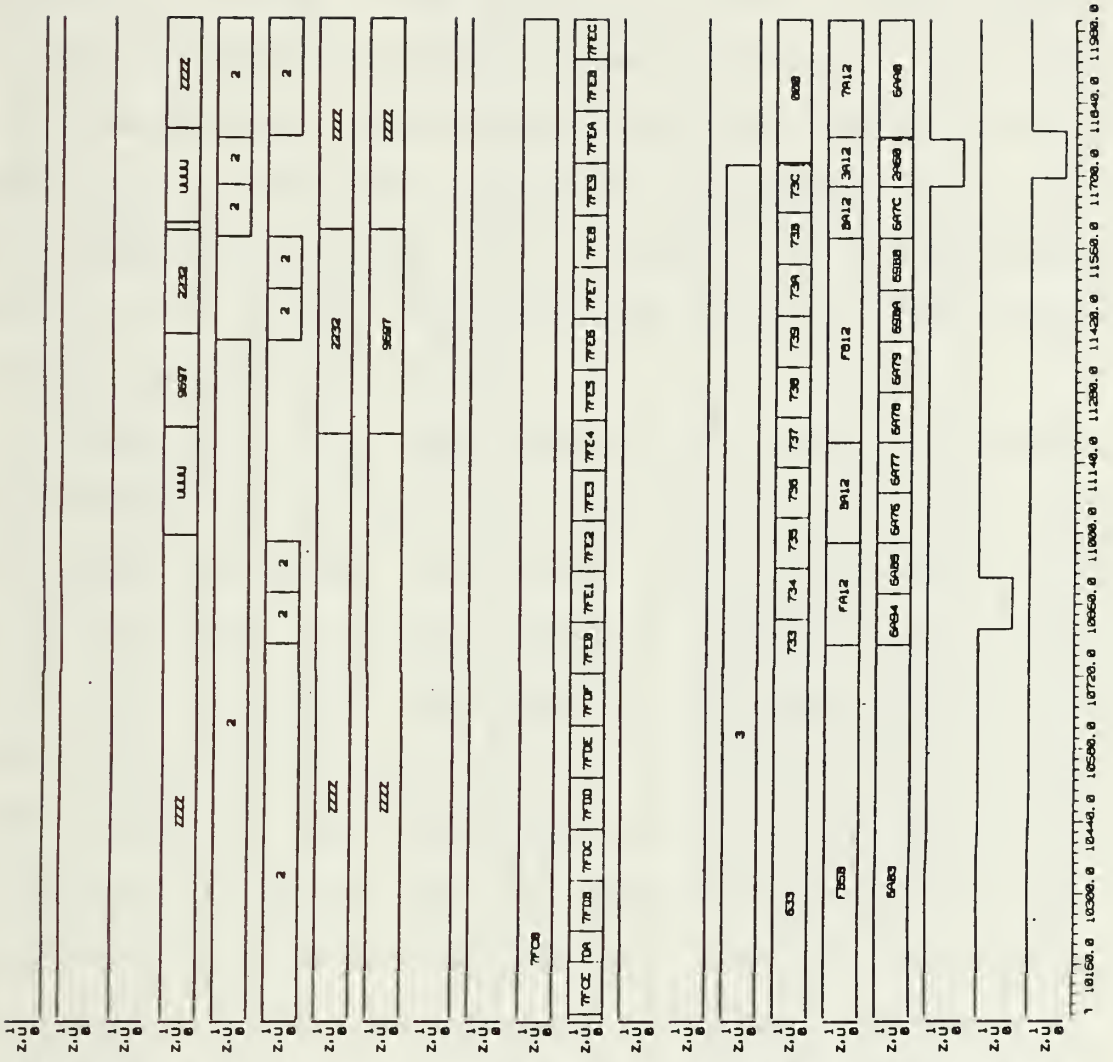


Figure 5-10: Computer Simulation for Instruction 011 (Cont)

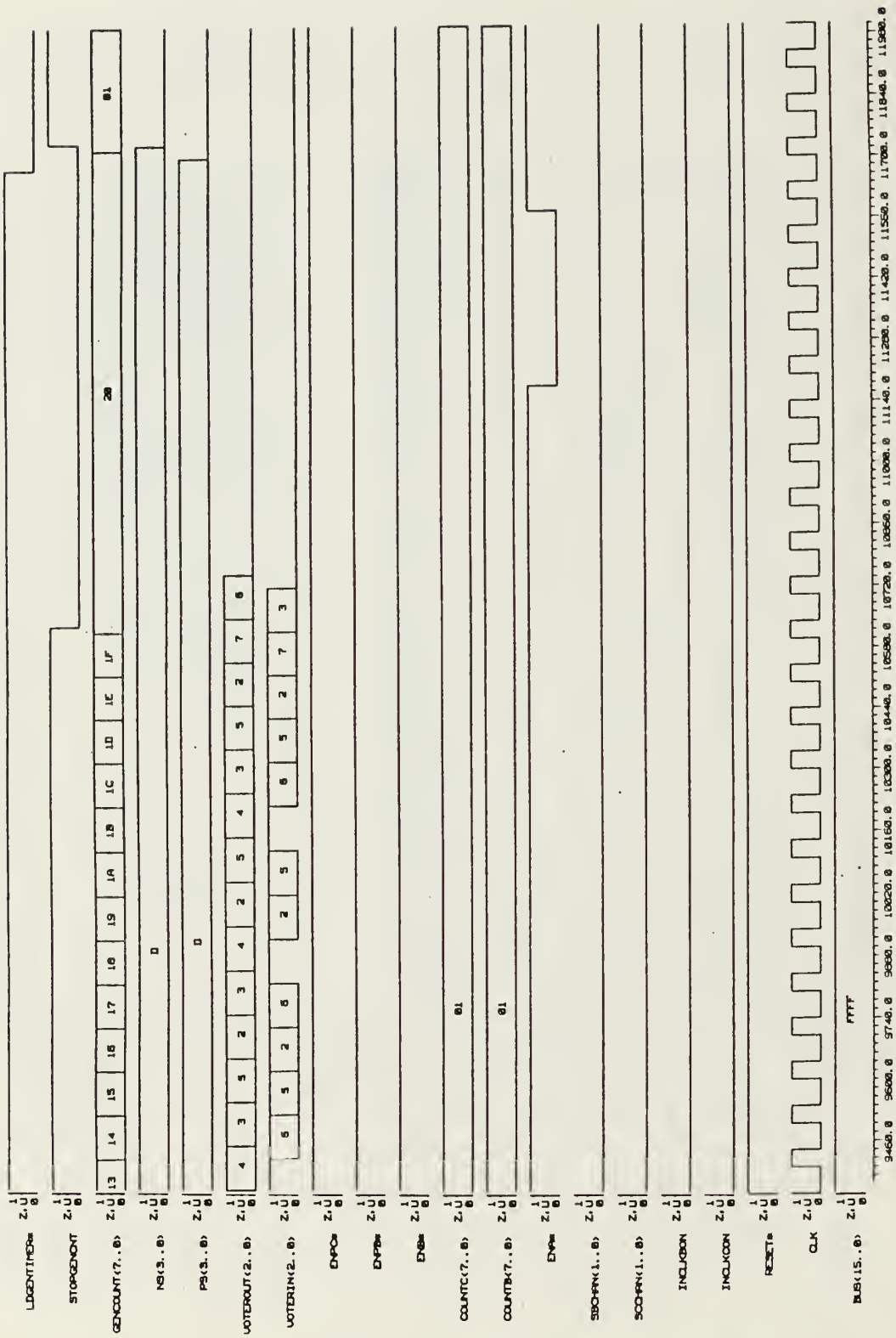


Figure 5-11: Computer Simulation for Instruction 011 (Cont)

H. PARALLEL TRANSFER A REG →CPU (INSTRUCTION 100)

This instruction transfers the contents of the A register to the CPU. The instruction takes 15 clock cycles. The CPU will send a doubleword operand which will be ignored by the interstage. However, the interstage must wait for at least eight clock cycles before it can transfer data to the CPU. After eight cycles, Q<5> is pulsed low and during the next two T1T4 cycles, the interstage transfers the contents of the A register to the CPU.

Figure 5-12 shows the computer simulation. At 3400 ns, SLVTOCPUSPC* is pulsed low signaling the CPU that the interstage is ready to transfer data. During the T1T4 cycle from 3600 - 3800 ns, the CPU is reading the status word. The interstage sets BUFSSR* low sending 00h as the status word. During the next two consecutive T1/T4 cycles, 3800 - 4200 ns, the interstage 74LS245 buffers enable a transfer to the CPU and the value in the A register, ABCD139Fh, is visible on the MBUSHI<15..0> bus. At 4200 ns, Q<14> pulses low and the interstage and CPU break contact.

I. LOAD CPU - A REGISTER (INSTRUCTION 101)

This is a five cycle sequence. Figure 5-13 shows the timing diagram. After decoding the instruction, the controller places the low word on the CPU's 16-bit external bus during the first T1/T4 cycle (800 - 1000 ns) and places

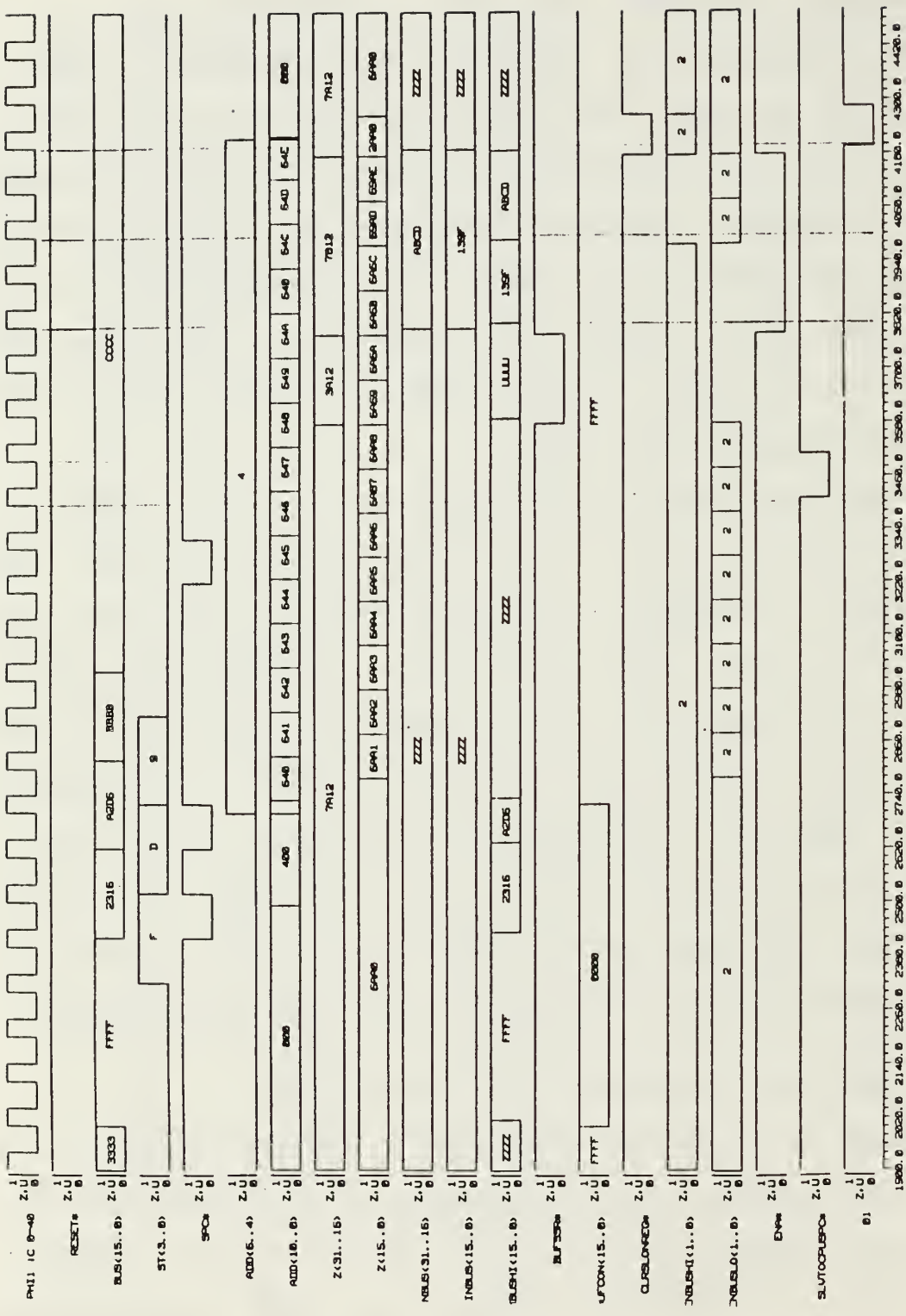


Figure 5-12: Computer Simulation for Instruction 100

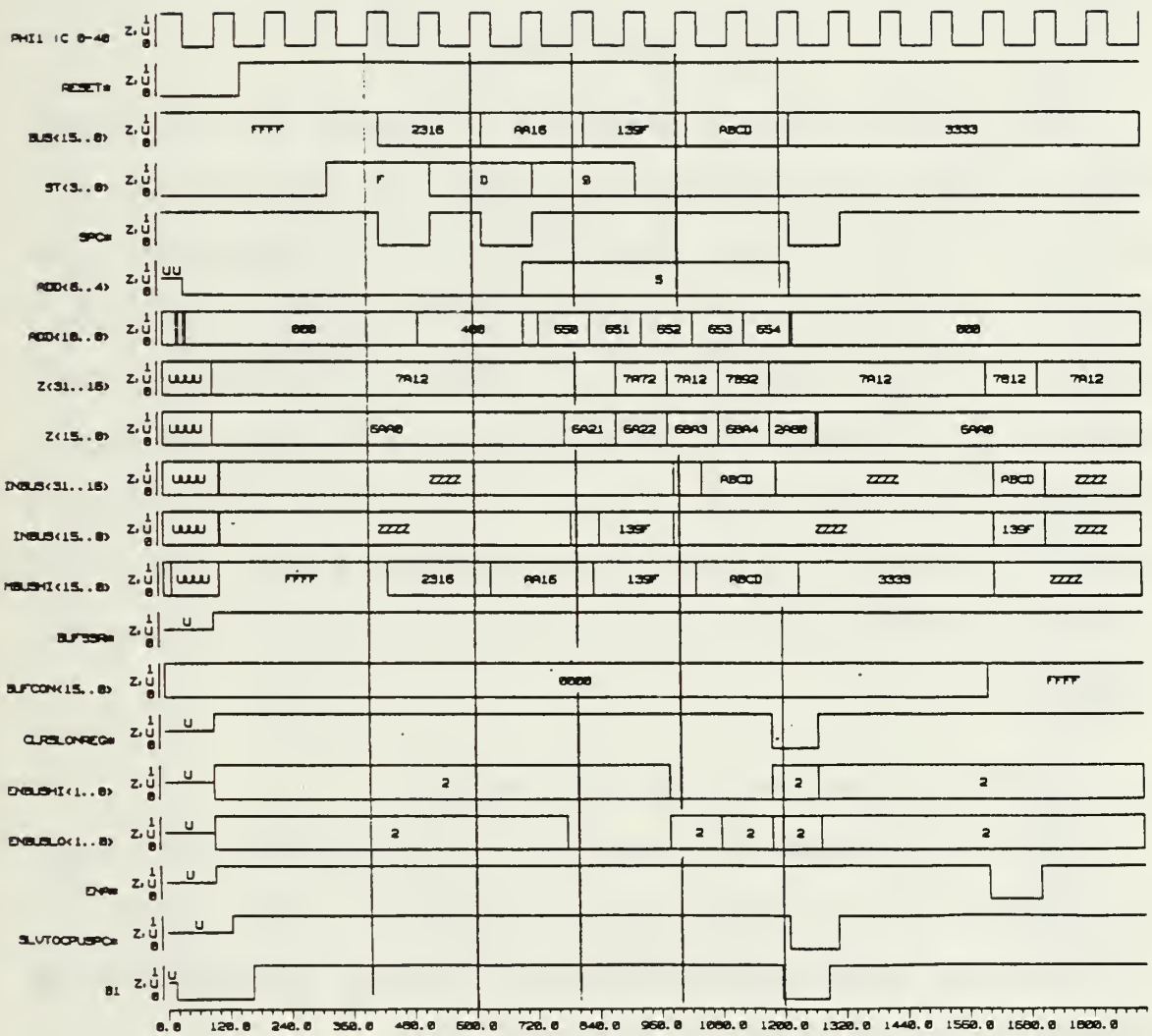


Figure 5-13: Computer Simulation for Instruction 101

the high word on the bus during the subsequent T1/T4 cycle (1000 - 1200 ns). MBUSHI<15..0> shows 139F and ABCD on the bus respectively. During the third T1/T4 cycle SLVTOCPUSPC* pulses low, Q<14> pulses low, and the controller returns to the wait state.

J. LOAD C REG → A REG (INSTRUCTION 110)

This instruction is identical in nature to instruction 000. It takes only three cycles and does not transfer any data to or from the CPU. The C register is enabled and the select lines for the A register are set for parallel loading. Figure 5-14 shows the computer simulation. The A register is loaded with CAFECAFEh (1340 - 1540 ns). After the load, Q<5> and Q<14> pulse low and the instruction is complete. ENA* is pulsed low at 3340-3540 ns to check the transfer. INBUS<31..0> during this time period verifies the transfer.

K. LOAD B REG → A REG (INSTRUCTION 111)

This instruction is identical to instruction 110 except that B is the source register instead of C. The B register is initially loaded with FACEFACEh. Figure 5-15 shows the computer simulation for this instruction.

L. SERIAL TRANSFER IN FROM EXTERNAL SOURCES

Serial loading the B' and C' registers from external interstages does not require a command from the CPU. Upon

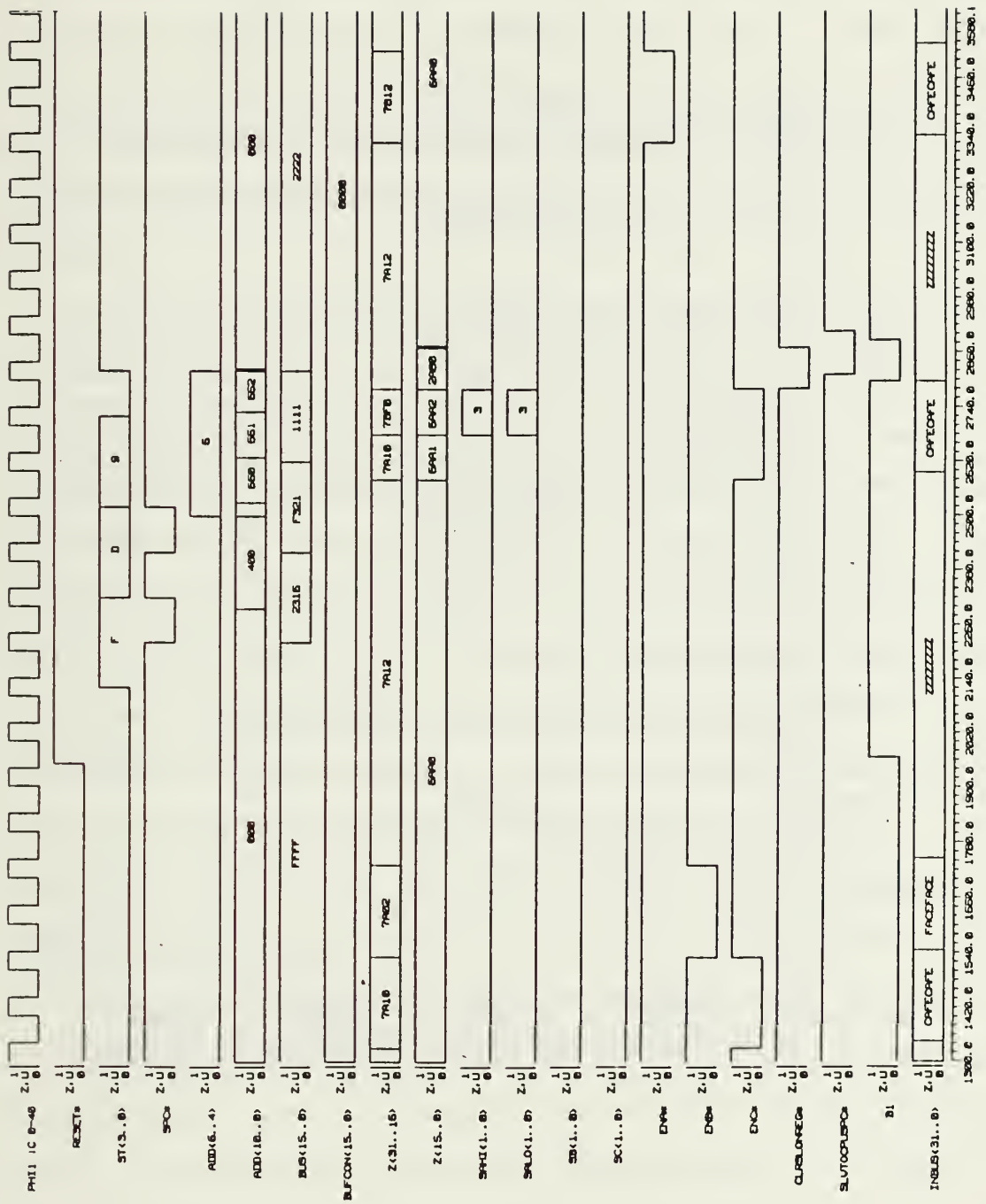


Figure 5-14: Computer Simulation of Instruction 110

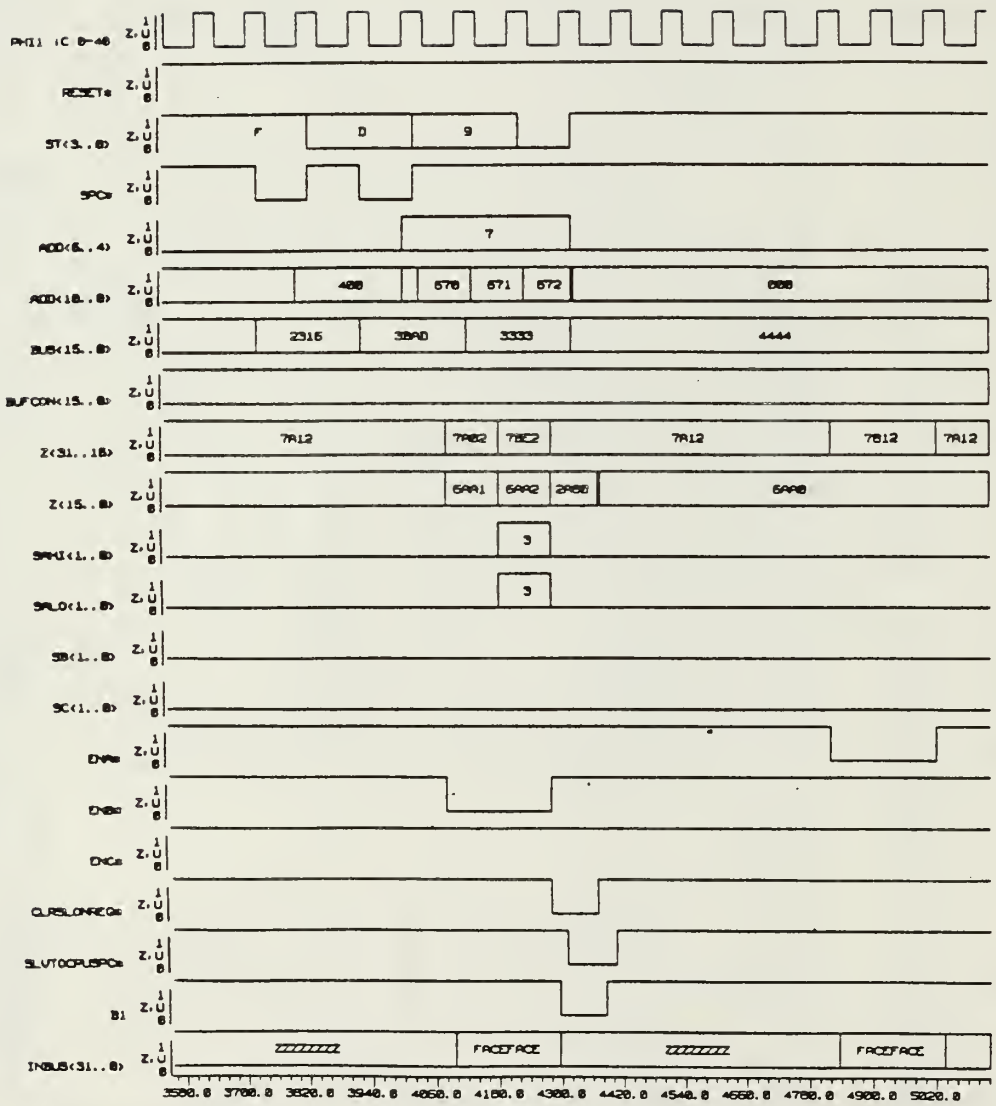


Figure 5-15: Computer Simulation for Instruction 111

a system RESET, the controller sets the select lines on the two registers to shift left. When the external interstage is ready to send data, it enables the clock. This clock signal is used to shift left data into the register and to count the number of shifts. After 32 bits have been received, the counter is disabled and the output of COUNT<5> is high.

Figures 5-16 and 5-17 show the computer simulation. At 2440 ns the interstage is in the process of executing instruction 010 when the external interstage enables both the B' and C' clocks and starts the serial load operation. The COUNTB<7..0> and the COUNTC<7..0> lines show the count progress. At 3600 ns, instruction 101, "load the A register from the CPU", begins. This does not affect the loading of the B' and C' registers. At 4360 ns, instruction 101 is complete. At 5600 ns, the external interstage disables the clock and the interstage counter suspends the count. The interstage is now in the wait state, but the B' and C' registers are loaded.

In this simulation, the A register was loaded in anticipation of a VOTE. In the VOTE simulation (Figures 5-8 through 5-11), the contents of the B' and C' registers were copied to the B and C registers respectively. At 7600 ns, INBUS<31..0> (Figure 5-8) shows the contents of the B' and C' registers. This verifies the successful loading of the B' and C' registers from external sources.

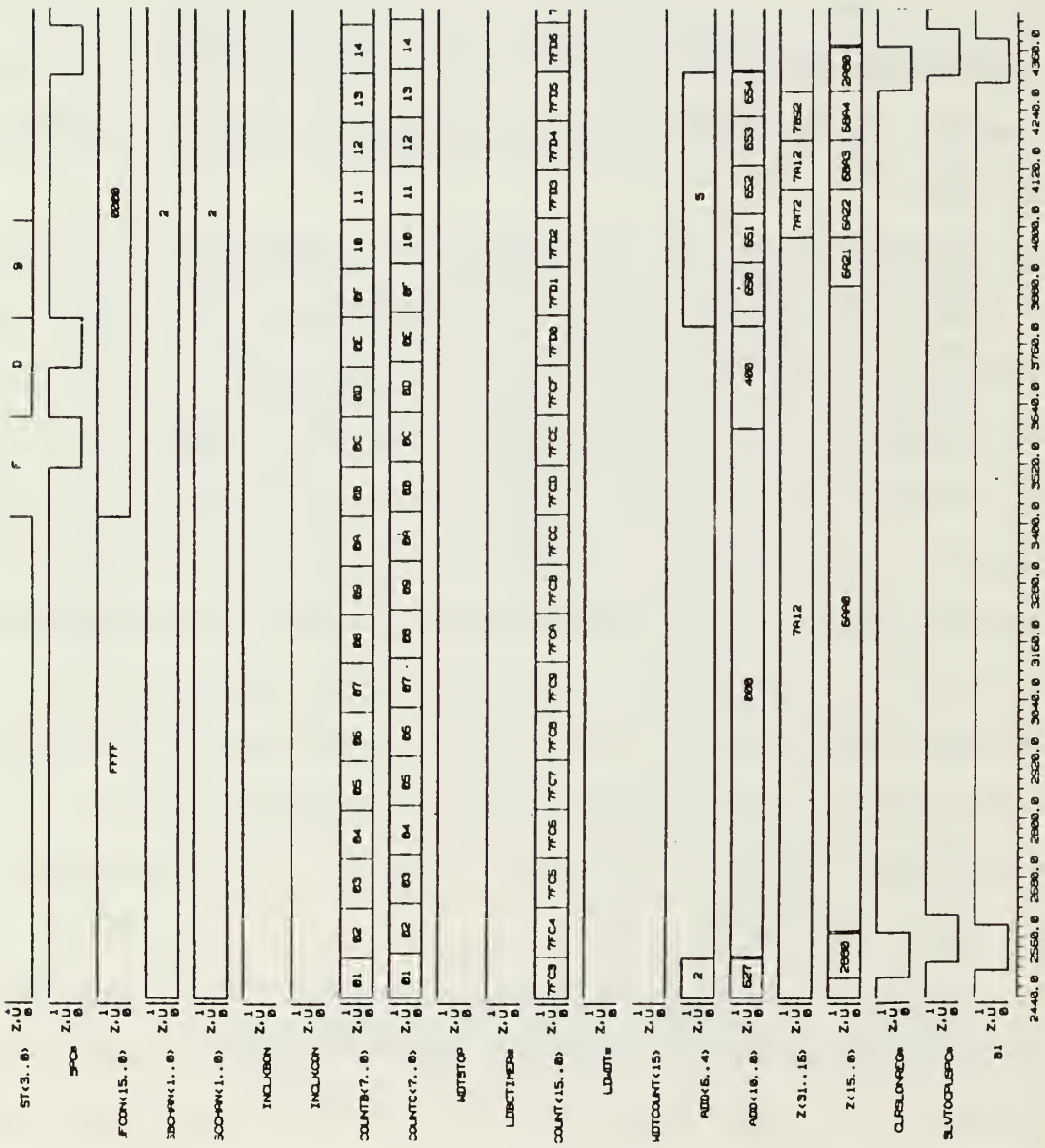


Figure 5-16: Computer Simulation for Serial Load of the B' and C' Registers

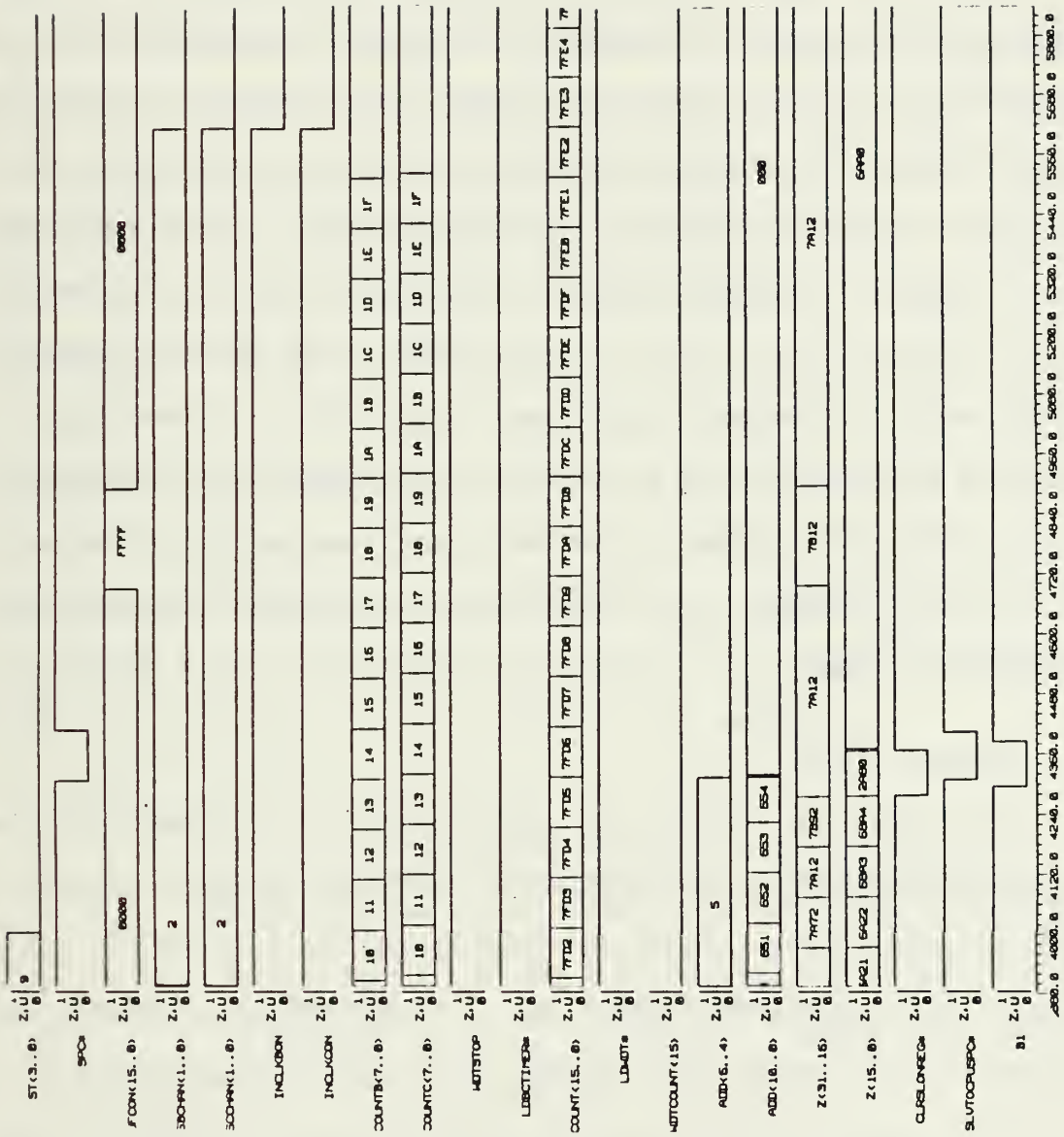


Figure 5-17: Computer Simulation for Serial Load of the B' and C' Registers (Cont)

VI. VOTER

The mid-value voter is designed as a finite state machine with seven variables. The voter serially inputs three 32-bit numbers and compares one bit from each clock cycle. Data is received from the A, B, and C 32-bit shift registers in the interstage by a shift-left operation. The voter output is concurrently left-shifted back into the shift registers with the middle value into A, the maximum value into B, and the minimum value into C. The middle value is shifted into A so it can be immediately transferred to the CPU for further comparison and processing. Parallel voting was examined, but was much too complex and hardware intensive to use.

A. INTEGER VOTE

Once the A, B, and C registers are loaded and tne "VOTE" instruction is received, the most significant bit (31) of each data field is presented to the voter. The voter will determine equality or inequality of the bits, determine the proper next state, then output the next bit of the mid, max and min values. On the leading edge of the next clock pulse, the voter output is left-shifted into the least significant bits (Ø) of each of the 32-bit shift registers and the next significant bits of each data field (3Ø) are presented to the voter. At the end of 32 clock

cycles, the registers contain the same numbers, but they are stored by mid, max, and min values in the A, B, and C registers respectively.

Operation of the voter can best be explained by using an example. Assume for simplicity that a 4-bit number is used and that register A contains 1001 (9), register B contains 0111 (7), and register C contains 1110 (14). By inspection, we can determine that the middle value is 9, the maximum 14, and the minimum 7.

Figure 6-1 shows the initial condition. The voter receives the first three bits, S3, compares them and determines that B is the smallest, and A and C are equal. The output of the voter is then MID = 1, MAX = 1, MIN = 0 and these bits are presented to the A, B, and C registers. They are shifted in on the first clock pulse.

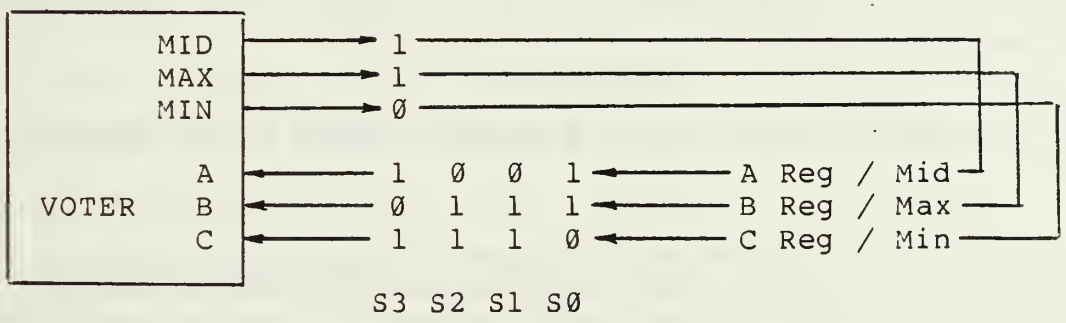


Figure 6-1: Voter Example: First Clock Pulse

Figure 6-2 shows the representation for the second clock pulse. The voter knows that B is min and the bit is automatically routed to MIN. Now C is determined to be

greater than A. The bit order is A = MID, C = MAX, B = MIN. This is a final state. No more decisions will be made and the voter will simply route the remaining input bits to their proper output channel. Figures 6-3 and 6-4 complete the voting cycle.

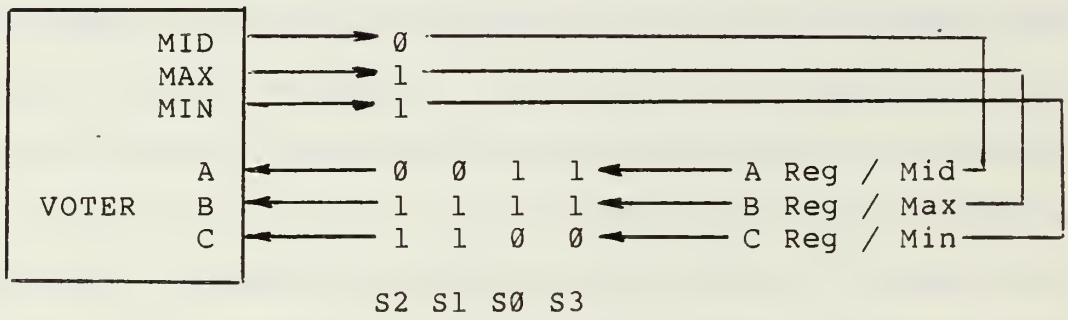


Figure 6-2: Voter Example: Second Clock Pulse

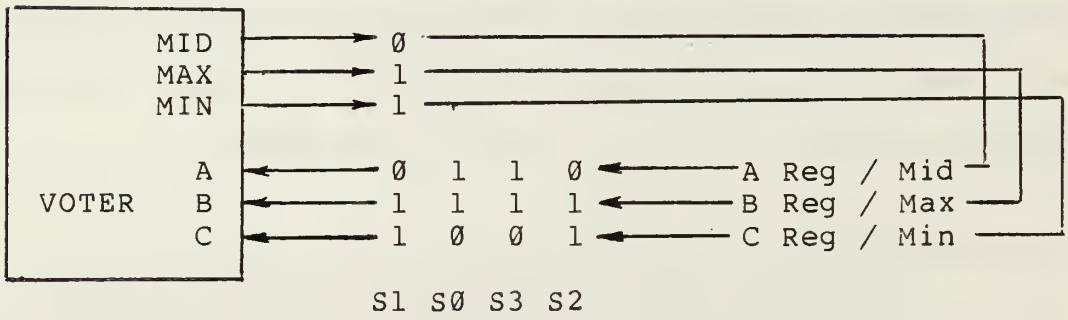


Figure 6-3: Voter Example: Third Clock Pulse

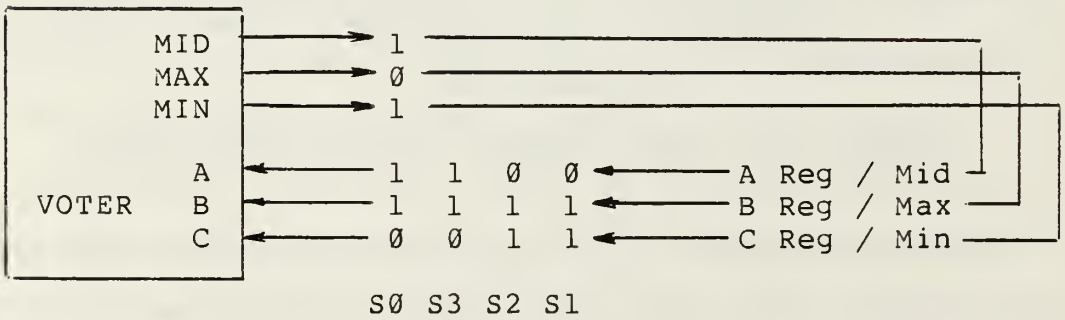


Figure 6-4: Voter Example: Fourth Clock Pulse

From Figure 6-4, the fourth clock pulse will load the shift registers as shown below:

```
1  Ø  Ø  1        A Reg / Mid
1  1  1  Ø        B Reg / Max
Ø  1  1  1        C Reg / Min

S3 S2 S1 SØ
```

The middle value, stored in register A, is 9; the maximum, 14, is in B; and the minimum, 7, is in C. At this point, the clock to the voter is disabled and the select lines for the 4-bit shift registers will prohibit any further shift operations. The values in the registers are stable.

To perform the vote, thirteen states are required. The state diagram is shown in Figure 6-5, the state table in Table 6-1, and the state assignments in Table 6-2. Four bits are required for the state variables and three bits are required for the inputs. With seven variables, a Karnaugh map cannot be used for minterm reduction. A Quine-McCluskey minterm reduction algorithm was written (Appendix B) and was used to determine the next state outputs.

The size and number of terms after the reduction is a function of state assignment. The state assignments shown in Table 6-2 (a) produce an output with nine terms. By comparison, the state assignments in Table 6-2 (b) produces an output with twelve terms. There is an optimal state assignment that will produce the smallest number of terms.

With 13 states, there are nine billion possible combinations and therefore no easy solution. The state

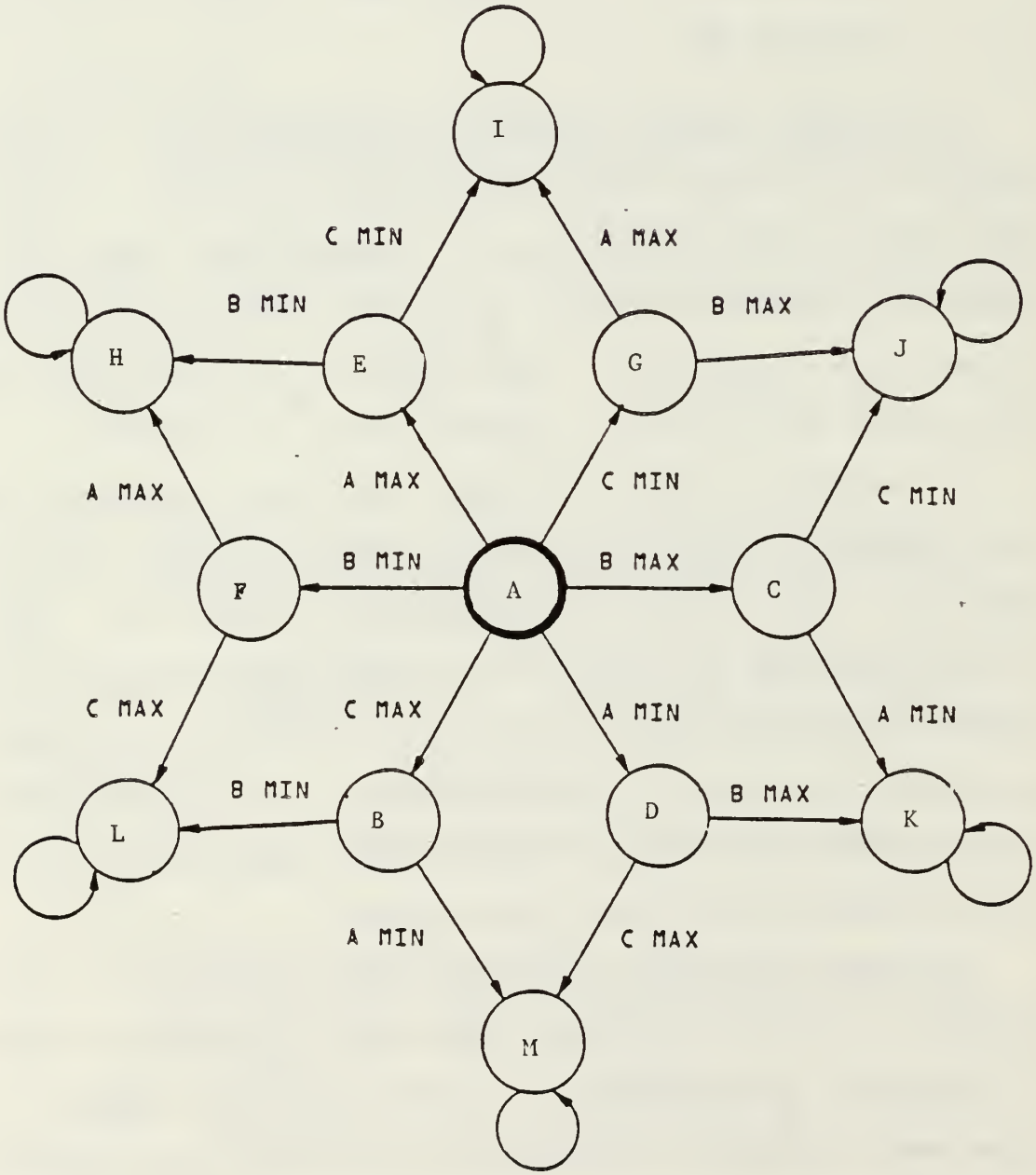


Figure 6-5: State Diagram for Mid-Value Voter

TABLE 6-1
STATE TABLE FOR MID-VALUE VOTER

A	B	C	Present State mx1	mx0	mn1	mn0	Next State MX1	MX0	MN1	MN0	Max	Mid	Min
0	0	0	A				A				X	X	X
0	0	1					B				C	X	X
0	1	0					C				B	X	X
0	1	1					D				X	X	A
1	0	0					E				A	X	X
1	0	1					F				X	X	B
1	1	0					G				X	X	C
1	1	1					A				X	X	X
0	0	X	B				B				C	X	X
0	1	X					M				C	B	A
1	0	X					L				C	A	B
1	1	X					B				C	X	X
0	X	0	C				C				B	X	X
0	X	1					K				B	C	A
1	X	0					J				B	A	C
1	X	1					C				B	X	X
X	0	0	E				E				A	X	X
X	0	1					H				A	C	B
X	1	0					I				A	B	C
X	1	1					E				A	X	X
X	0	0	D				D				X	X	A
X	0	1					M				C	B	A
X	1	0					K				B	C	A
X	1	1					D				X	X	A
0	X	0	F				F				X	X	B
0	X	1					L				C	A	B
1	X	0					H				A	C	B
1	X	1					F				X	X	B
0	0	X	G				G				X	X	C
0	1	X					J				B	A	C
1	0	X					I				A	B	C
1	1	X					G				X	X	C
X	X	X	H				H				A	C	B
X	X	X	I				I				A	B	C
X	X	X	J				J				B	A	C
X	X	X	K				K				B	C	A
X	X	X	L				L				C	A	B
X	X	X	M				M				C	B	A

assignment in Table 6-2 (a) was used to design the voter since it had the fewest terms.

TABLE 6-2
STATE ASSIGNMENTS

	(a)		(b)
A	0000	A	0000
B	0100	B	0101
C	1000	C	0110
D	1100	D	1010
E	0001	E	1001
F	0010	F	0111
G	0011	G	1110
H	0110	H	0011
I	0111	I	0001
J	1001	J	0010
K	1011	K	0100
L	1110	L	1000
M	1101	M	1100

For ease in using the Quine-McCluskey algorithm, the 4-bit present state variables were labeled as D, E, F, and G. The next state variables were labeled as MX1, MX0, MN1 and MN0. The state variable table in Table 6-1 now looks like:

A	B	C	Present State				Next State			
A	B	C	D	E	F	G	MX1	MX0	MN1	MN0
0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	0	0
0	1	0	0	0	0	0	1	0	0	0
.		.	.			.	.			.
.		.	.			.	.			.
.		.	.			.	.			.

By using the state assignment in Table 6-2, the Quine-McCluskey algorithm produced the following terms:

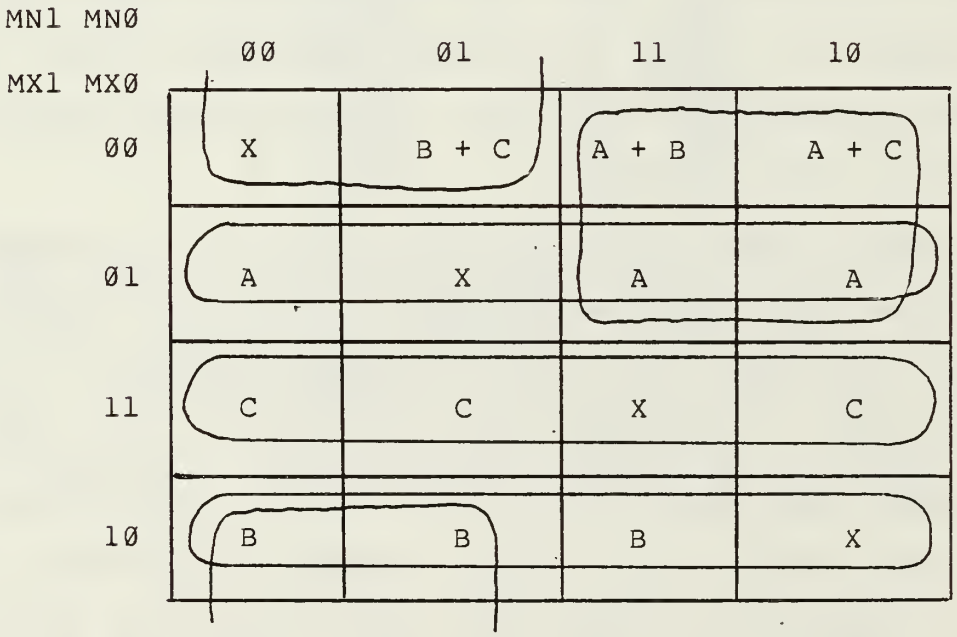
```
MX1 = A*CD*E*FG* + A*B*CE*F* + A*BC*E*F* + B*CE*F*G
      + BC*E*F*G + A*BE*FG + DE*G + DEG* + DF*

MXØ = B*CD*E*F*G + A*B*CD*G* + AB*C*D*G* + A*CD*FG*
      + AC*D*FG* + AB*D*FG + D*EF + DEF* + EF*

MN1 = AC*DE*F*G* + ABC*E*F*G* + B*CD*EG* + BC*D*EG*
      + AB*CD*G* + AB*DEG* + E*FG + EFG* + E*G

MNØ = BC*D*EF*G* + ABC*D*F*G* + A*CDE*F* + A*BCE*F*
      + AC*DE*F* + A*BDEF* + D*FG + DF*G + E*G
```

Appendix B shows that MX1, MXØ, MN1, and MNØ each reduce from a 58 term canonical sum to a 9 term sum of products expression. This implementation requires ten gates per output for a total of 4Ø gates (see Figure 6-1Ø). Using the same procedure for MAX, MID, and MIN should also require a similar number of gates.

However, better minterm reduction can be achieved by using the next state values MX1, MXØ, MN1, and MNØ as inputs. The method used is not a standard approach. Instead of filling a Karnaugh map with "1's" (for minterm reduction), the map is filled with logical equivalents from the state table. For example, to compute the MAX value, refer to Table 6-1, Table 6-2, and Figure 6-6. The first block in the Karnaugh map, ØØØØ is a don't care. This means that if the state of the voter is ØØØØ, the positions of the max, mid, and min outputs do not matter since they are equal. The next block in the map, ØØØ1, is the state where A has been determined to be the min value. Therefore, the max value can be either B or C since they are equal. This

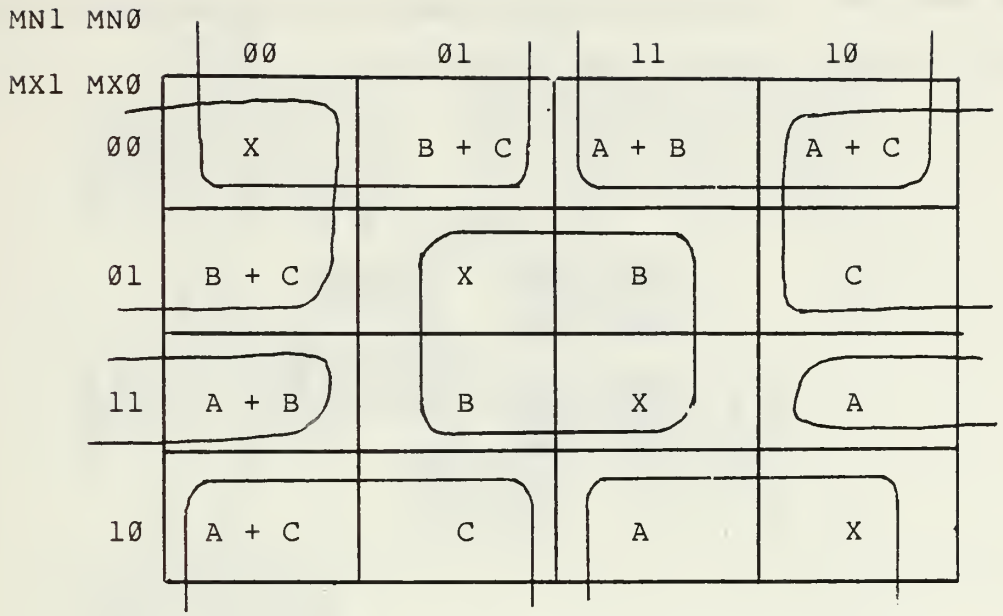
same logic is used to complete the remainder of the Karnaugh maps. Figures 6-6, 6-7, and 6-8 compute MAX, MID, and MIN respectively.

The logic diagram for MAX, MID, and MIN is shown in Figure 6-10. The hierarchial body for the voter is shown in Figure 6-9. In computing the three voter outputs, hardware has been reduced from 30 gates to 23 gates. Although seven gates does not seem like much, the size of the gates used is



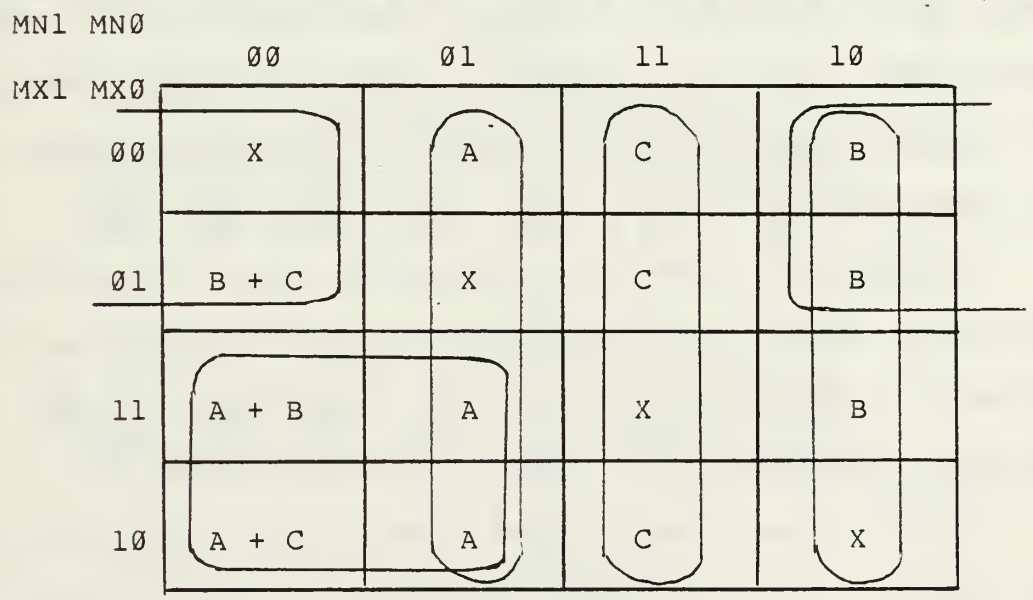
MAX = MX1 MX0* B + MX1 MX0 C + MX1* MX0 A +
MX1* MN1 A + MX0* MN1* B

Figure 6-6: Computation of MAX



MID = MX0* MN1 A + MX0 MN0 B + MX0* MN1* C +
MX1* MN1* B + MX1 MX0 MN0* A + MX1* MN0* C

Figure 6-7: Computation of MID



MIN = MN1* MN0 A + MN1 MN0 C + MN1 MN0* C +
MN1* MN0* A + MX1 MN1* A

Figure 6-8: Computation of MIN

much smaller. Simpler gates equate to higher reliability, which is the ultimate goal of the design.

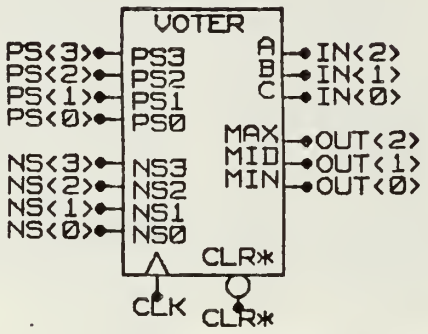


Figure 6-9: Hierarchial Body for the Voter

The voter is designed with LSTTL components since the SCALD CAD machine has only a complete library for LSTTL components. However, a severe speed penalty is paid by using LSTTL. Table 6-3 compares speeds for LSTTL, STTL, and FAST chips [Refs. 11, 12]. Only the chips used in the voter are compared.

Propagation times for each chip type is compared in Table 6-4. Best case, worst case and maximum and minimum propagation delays are computed. The SCALD system uses nominal propagation delay times for simulations.

It is obvious that a voter constructed with LSTTL chips will not work with a 10 MHz clock. Of the six test simulations shown in Figures 6-11, 6-12, and 6-13, none provide the correct max, mid, and min outputs using a 10 MHz

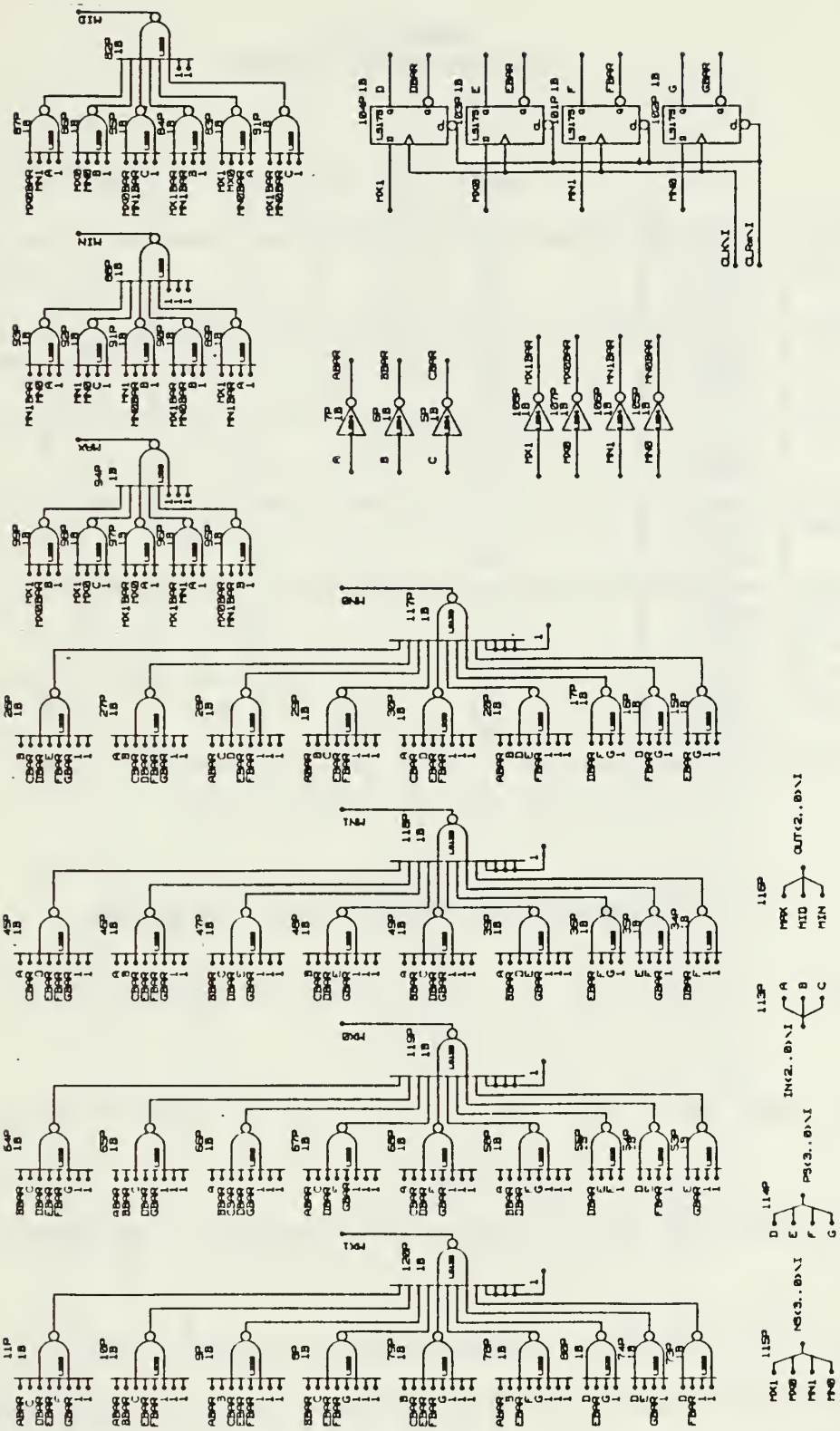


Figure 6-10: Voter Schematic

TABLE 6-3
PROPAGATION DELAYS

		LS		S		FAST	
		MIN	MAX	MIN	MAX	MIN	MAX
74 '133	tPLH		15		6	No data	
	tPHL		38		7		
74 '30	tPLH		12	2	6	No data	
	tPHL		20	2	7		
74 '20	tPLH		15	2	4.5	2.4	6
	tPHL		15	2	5	2	5.3
74 '04	tPLH		10	2	4.5	2.4	6
	tPHL		10	2	5	2	5.3
74 '175	tPLH		25		12	4	7.5
CP to Q	tPHL		25		17	4	9.5
74 '00	tPLH		10	2	4.5	2.4	6
	tPHL		10	2	5	2	5.3

* All times are in nanoseconds

Table 6-4
PROPAGATION TIMES THROUGH THE VOTER

GATES '04 + '30 + '133 + '175 + '20 + '30 + '04 =

Using LSTTL

Worst (MAX) 10 + 20 + 38 + 25 + 15 + 20 + 10 = 138 ns

Using STTL

Worst (MAX) 5 + 7 + 7 + 17 + 5 + 7 + 5 = 53 ns

clock. The failure is caused by the 3-level circuit used to compute max, mid, and min (Figure 6-10). With the clock slowed to 5 MHz, the voter worked perfectly. The voter

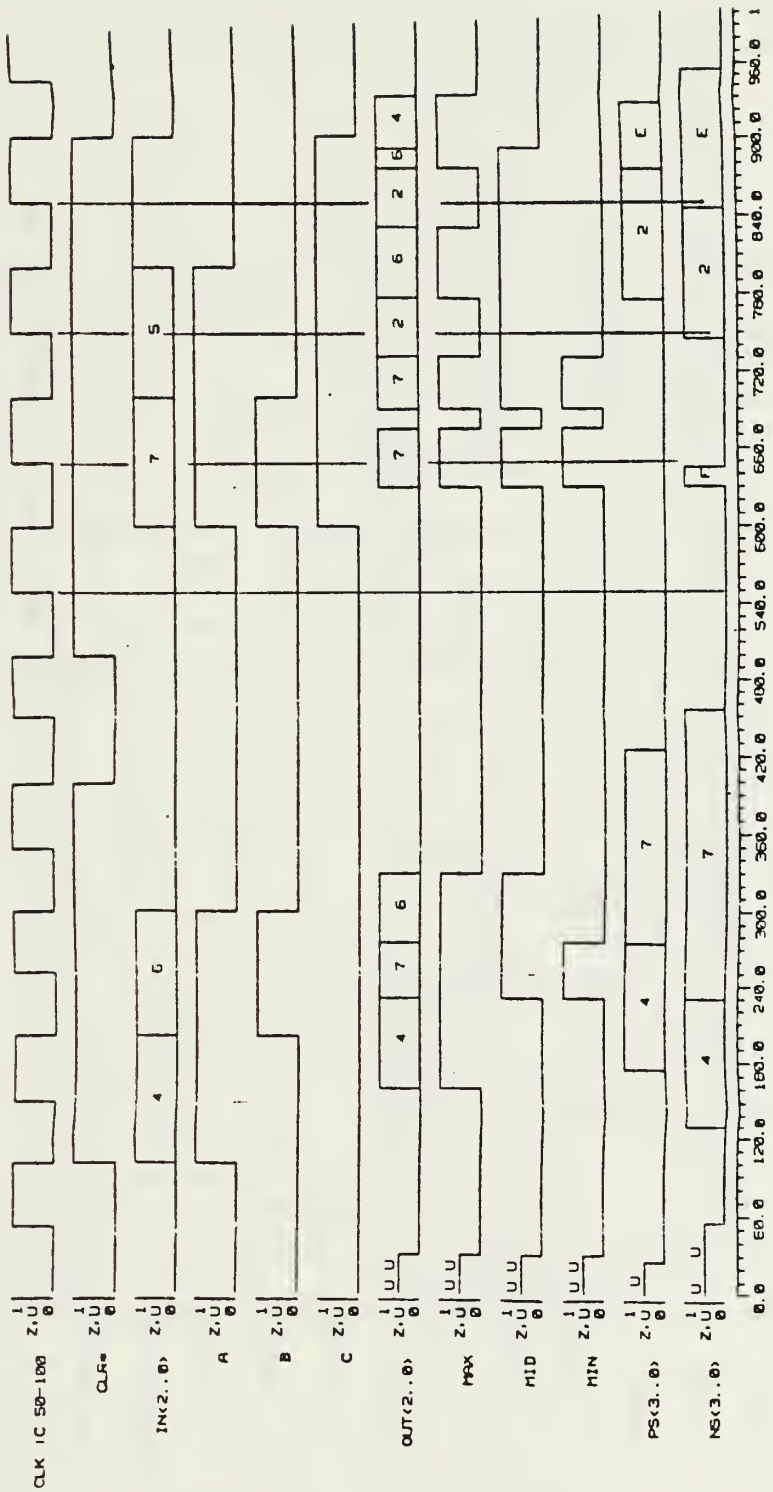


Figure 6-11: Computer Simulation for the Voter (10 MHz)

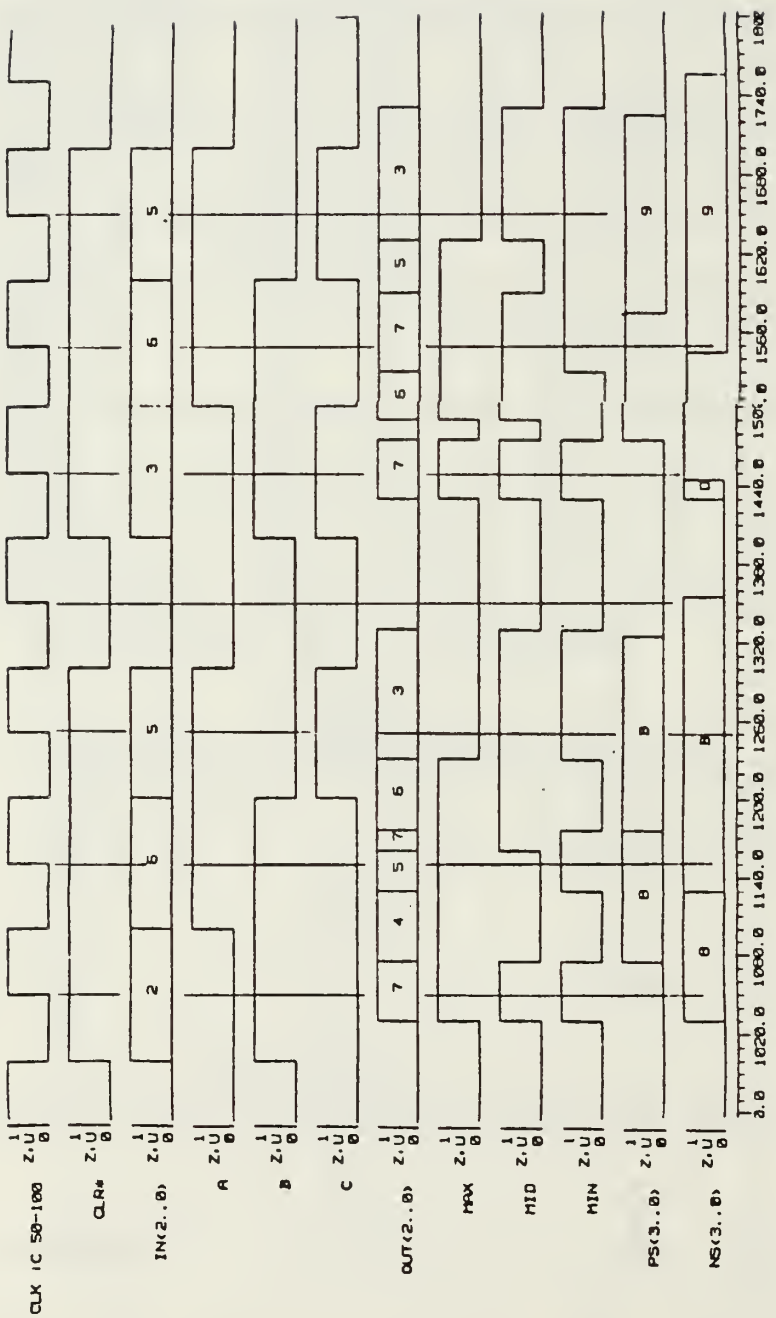


Figure 6-12: Computer Simulation for the Voter (10 MHz)

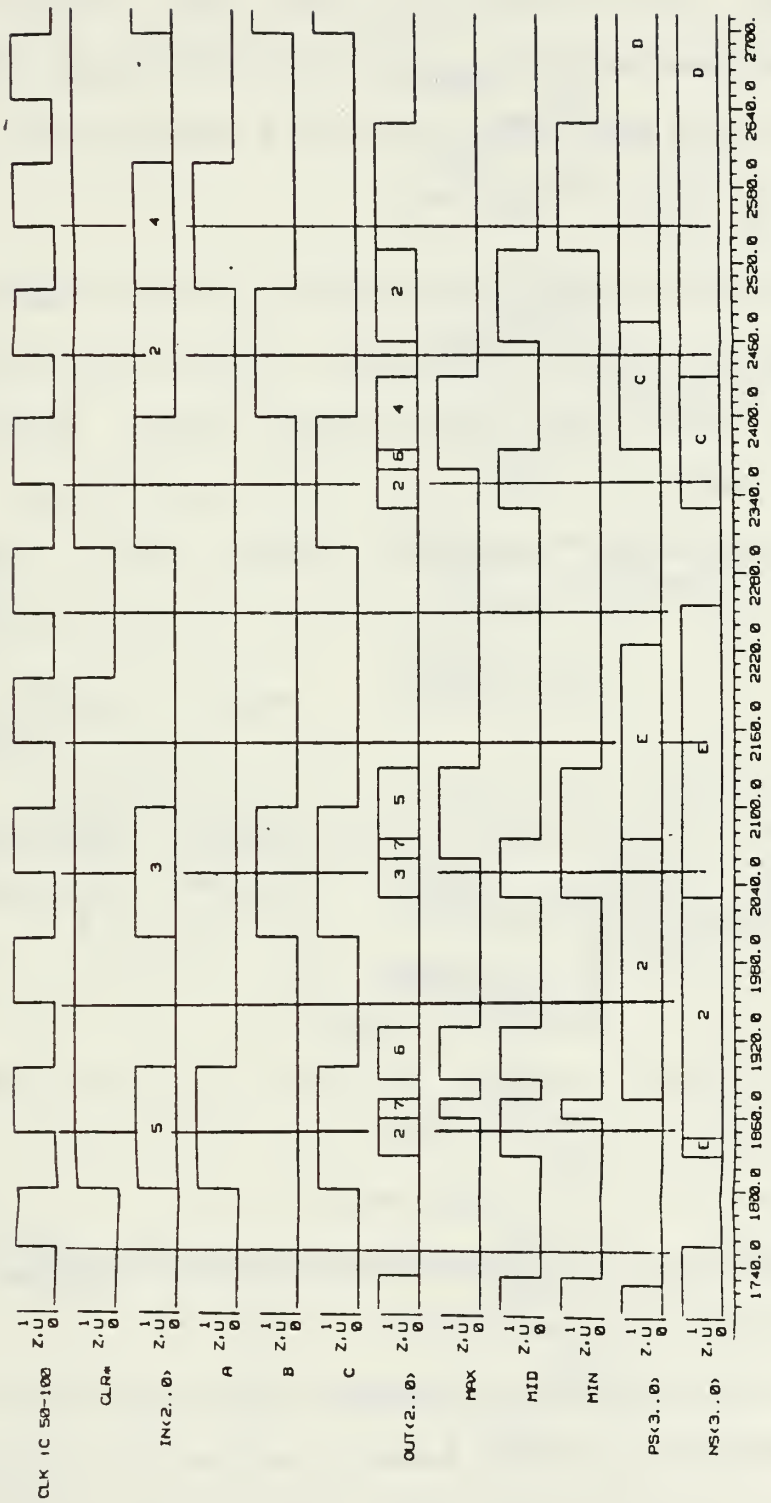


Figure 6-13: Computer Simulation for the Voter (10 MHz)

design works, but not with a 10 MHz clock. Table 6-4 shows that if the breadboard prototype is build using STTL chips, the voter should work properly with a 10MHz clock.

B. ALTERNATE VOTER

Figure 6-14 shows an alternate voter design. This design uses a 3-bit steering circuit and has more propagation delay than the original voter. However, the gate count and gate complexity is significantly reduced. Only three different gates are needed: '00, '04, '20; and a register, '175.

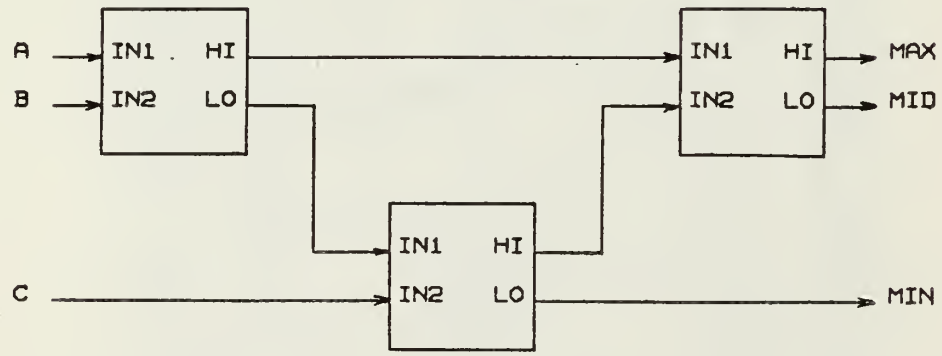


Figure 6-14: Alternate Voter Block Diagram

Using a Karnaugh map, a 15-gate implementation for each stage is developed (Figure 6-15). Two registers store the state of each stage (both equal, IN1 > IN2 or IN2 > IN1). Table 6-5 compares the two voters.

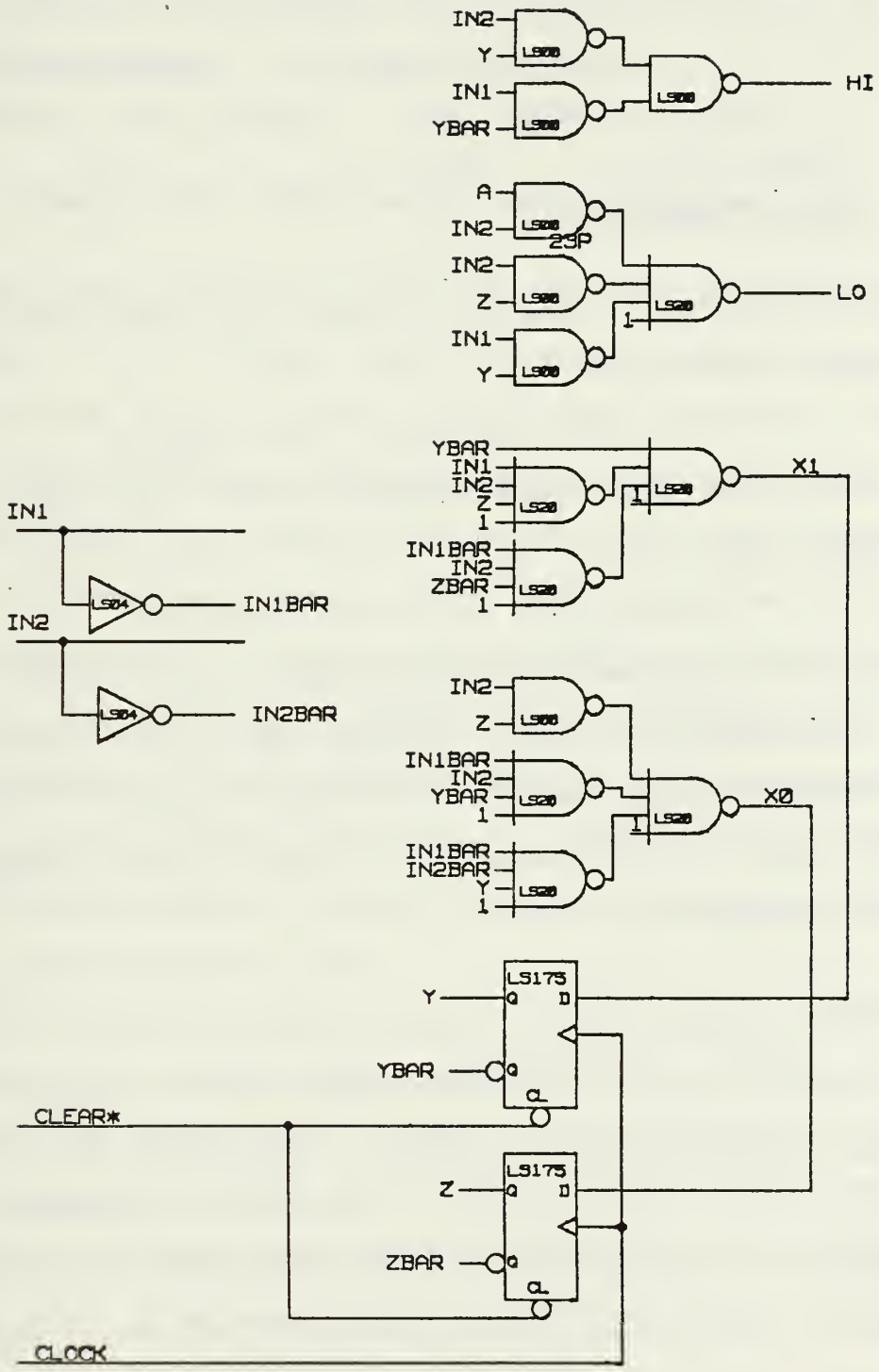


Figure 6-15: Alternate Voter Schematic (One Section)

TABLE 6-5
COMPARISON OF ORIGINAL AND ALTERNATE VOTER

	Original Voter	Alternate Voter
Number gates	66	45
Number registers	4	6

Using LSTTL for the alternate voter, the worst case propagation delay is

$$3 \times (15 + 10 + 10) + 25 = 140 \text{ ns}$$

Using STTL for the alternate voter, the worst case propagation delay is

$$3 \times (5 + 5 + 5) + 17 = 62 \text{ ns}$$

The STTL implementation would work with a 10 MHz clock. A VLSI implementation would further reduce the propagation time significantly and the alternate voter could be the primary voter in VLSI due to its simplicity and subsequent greater reliability.

C. FLOATING POINT VOTE

The voter was originally designed to vote either integer or floating point numbers. The format for the IEEE floating point standard [Ref 9] is shown in Figure 6-16. The mantissa is normalized with the most significant "1" not present in the 23-bit mantissa representation.

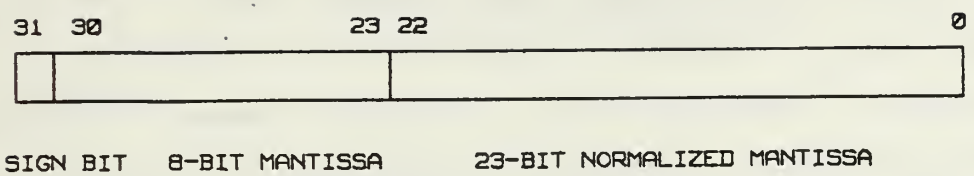


Figure 6-16: IEEE Floating Point Representation

The IEEE floating point representation could use the integer voter except for the sign bit in the most significant bit. If the sign bit was inverted, then the voter would perform an accurate vote. Figure 6-17 shows how six XOR gates could perform this function. The signal "FP" would be high only long enough for the first bit to be inverted prior to entering the voter and inverted again prior to being stored in the register. This concept could not successfully be simulated since it only aggravated the timing problem discussed earlier. Use of the XOR gates is the simplest method to switch between integer and floating point numbers for the vote.

Use of the XOR gates create another problem. The gates work only for certain combinations of signed numbers. For example, if the numbers 5, -7, and -4 are voted, the max-mid-min output should be 5, -4, -7. The serial voter is designed as a magnitude voter and its output for the example would be 5, -7, -4, which is clearly incorrect. To design the voter to recognize and respond to this condition

requires more hardware which is contrary to increasing reliability. Floating point votes are therefore too difficult to implement in a simple manner. Hardware floating point voting should be reexamined in light of the above to determine whether it is feasible or not.

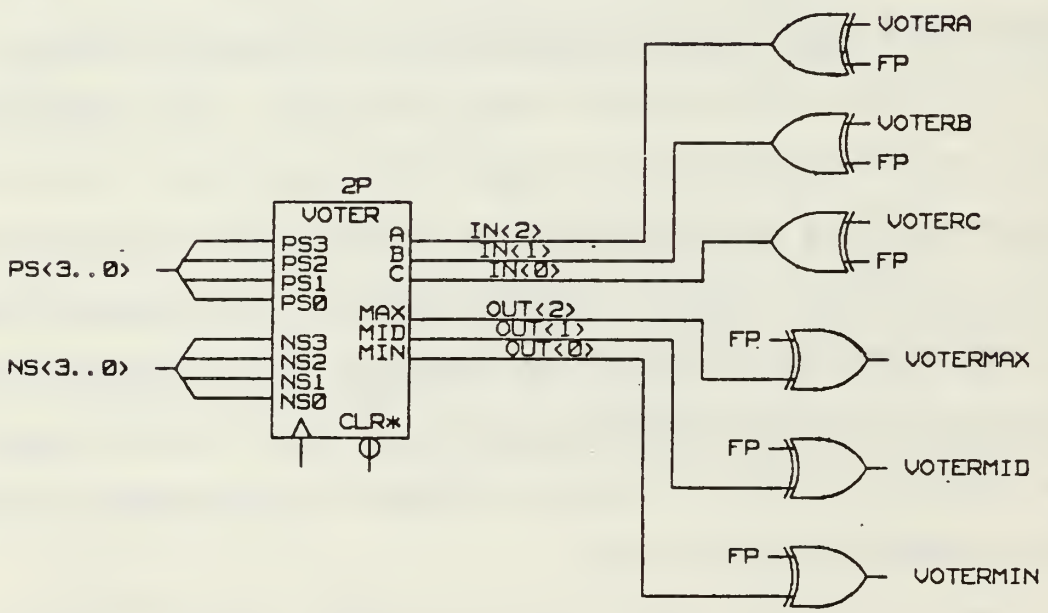


Figure 6-17: Floating Point Vote

VII. SUMMARY AND CONCLUSIONS

A. SUMMARY

The original goal of the thesis was three-fold. First, design a voter, an interstage, and examine procedures to interface the voter and interstage to the NS32Ø16-1Ø CPU. The second goal was to use the Valid SCALD CAD terminal at the Naval Postgraduate School to simulate and verify the design. The third involved determining reliability of the final design and wirewrapping a breadboard prototype.

The first item designed was the voter. This turned out to be more difficult that first appeared and consequently the Quine-McCluskey algorithm had to be written. The voter was successfully tested with both boolean logic programs written in basic and the SCALD CAD system. Restricted to LSTTL chips by the SCALD CAD system library, the voter could not be tested at 1Ø MHz but worked at slower speeds. Floating point voting was not successfully simulated due to the propagation delays associated with LSTTL chips.

One of the primary functions of the interstage is to serially exchange data from two external channels simultaneously. Full duplex operation requires four shift registers. A fifth shift register is used to read and write a doubleword to the CPU. In a typical application, the interstage can shift data in from the external channels and

simultaneously: a) read or write to the CPU, b) serially transfer data out, or c) vote. A 16-bit watchdog timer and three modulo-32 counters are used to control the shifts.

The first design iteration for the interstage used more than 32 control lines. This would require five 2K x 8 PROM's for the controller. The control lines were eventually reduced to 28 with four lines for the FSM states.

To keep the serial data exchange full duplex, the design process revealed that more than three mod-32 counters would be required or else more control lines to the external interstages (such as Data Sent and Data Acknowledged). This would require more decoding hardware and more control lines from the controller.

Interfacing the interstage to the NS32Ø16-1Ø CPU turned out to be very challenging. After many phone calls to National Semiconductor to clarify the explanations given in the databook, the control signals that the CPU sends to the interstage were simulated on the SCALD CAD system. The interstage then responded to these CPU commands and executed the given instruction.

Using the SCALD CAD machine turned out to be extremely time consuming and not as efficient as one would think. The operating system version on the machine when the thesis began was 7.51. Drawing compilations and simulations were extremely slow. For a circuit the size of the thesis, compilations and simulations took about an hour.

The SCALD CAD system was very complicated and difficult to learn to use. Reading the manuals was not a simple task. Also there were software bugs in the system. One particular simulation error with the buffers took over 30 hours (most due to slow simulation times) to isolate the problem which was subsequently acknowledged by Valid Inc. as a "known bug". An inordinate amount of time was then spent trying to simulate and model around the "known bugs" to validate the design. A lot of time was wasted trying to compensate for software deficiencies in the SCALD.

A new operating system, Version 8.0, was installed in February 1986. The problems with the buffer were resolved in the new version and the simulator was much faster. It was now feasible to simulate the entire instruction set and validate the complete design.

Although the simulator was extremely fast, it still took about 15 minutes to set-up and enter the simulator. Recovery from swap space errors, which were frequent, took about 20 minutes.

Another problem with the SCALD was the number of students using the machine for assigned laboratories. With two classes and thesis students, the machine's disk space was always near capacity. Although unverified, the SCALD machine seemed to have more problems when disk space used was over 95 percent.

The SCALD system simulations were very accurate and pointed out timing and synchronization problems. These simulations are responsible for the conclusions presented in this thesis. Using the simulator to find and eliminate design errors, according to experienced personnel, was much easier than troubleshooting a wirewrapped prototype. This is probably a very fair statement considering that the interstage has 32 control lines, four 8-bit counters, one 16-bit counter, a 32-bit internal bus, and five 32-bit shift registers.

The SCALD system is not very user friendly and is very difficult to learn to use. Swap space errors, which locked up the computer, were frequent and required a reboot in order to recover. Overall, the SCALD system is not recommended for use by an inexperienced person. The time required to learn the machine's capabilities offsets most of the gains that can be realized from the computer simulations.

The breadboard prototype was not constructed since a final design was not developed. The initial design effort revealed flaws in the original concept for the interstage and the actual design of the voter. Until the voter and interstage design is reexamined, there is no need to construct a prototype model.

B. CONCLUSIONS

The concept for the interstage in this thesis uses too much hardware. It is clearly too complex to be viable and the complexity of the interstage hurts the overall reliability. The interstage must be reduced in scope.

First, the mod-32 counters used to count the bits arriving from the external interstages are not necessary. All these counters are doing is counting clock pulses. If a noise spike appears on the external clock line, one or more bits of the 32-bit stream will be lost since the counter will count the spike as the first bit, then stop the counter one bit early. If the noise spike appears after the data is input and prior to the vote, then the counter provides protection against error. It definitely is not worth all the extra hardware to catch a 50% probability of error. Furthermore, it is unimportant to know whether the clock line or the data line is malfunctioning. When the SIR network shuts the malfunctioning computer network down, it does not care what the problem is since it cannot fix the problem. Maintenance personnel who remove the defective network will determine the exact nature of the failure.

The timing problem caused by using the GENTIMER to count the bits entering the voter and to count the bits transferred to external interstages can be solved by adding an extra counter. One counter would be dedicated to

counting during voting and the other would be dedicated to counting bits transferred out to the other interstages.

The B' and C' registers are necessary to provide data storage during full duplex operations. If the system the fault tolerant computer supports operates slow enough, full duplex operations may not be necessary. Elimination of this feature will remove two 32-bit shift registers and their associated control lines.

The watchdog timer register could be eliminated. For most applications, the watchdog timer will count for a fixed period of time, provide a flag, then will be reset. The start count could be hardwired to the counter and the watchdog timer would not be necessary. Removal of the watchdog timer register must be weighed against the need to provide a variable watchdog timer window.

More research should be done to decide whether a hardware voter is more reliable in the long run than a software voter. Unquestionably, the serial hardware voter is at least 3 to 4 times faster than a CPU. The hardware voter can perform a vote on 32-bit data in 32 clock cycles. A software voter would require much more time due to the overhead of fetching instructions, transferring and comparing vote results, and the fact that most CPU execution cycles take at least three of four clock cycles.

Also, the algorithms used to simulate the CPU on the SCALD system have to be verified. This requires physically stepping through the slave instructions provided by the NS 32016-10 CPU, watching the CPU control lines, and verifying the timing. If the test verifies the procedures used in the computer simulation, then the timing in the interstage controller is accurate.

Finally, using the Custom Slave Processor mode in the NS 32016-10 CPU works very well. Interfacing and passing instructions and operands is simple. The Custom Processor can be designed to act as a parallel processor with a minimum of hardware.

APPENDIX A: SCALD APPLICATION NOTES

The SCALD system, by Valid Logic Systems, Inc., is an interactive, high resolution graphics digital logic simulator. A keyboard and mouse are used to draw logic diagrams on a video screen. Text files, required for compilation and simulation, are supported by a UNIX operating system.

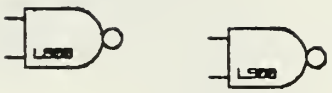
Libraries resident in the disk contain the components used in the simulations. Of concern to this thesis is the LSTTL and FAST libraries. The STTL library, which would have been used because of the small propagation delays, is currently not installed at the school.

After logging onto the system, the Graphics Editor (GED) is entered. The mouse controls a moving cursor on the screen. Components from the library are attached to the cursor and fixed into position (Figure B-1a). Wires are added to interconnect the bodies (Figure B-1b). Once the drawing is completed (Figure B-1c), signal names are attached to the wires (Figure B-1d).

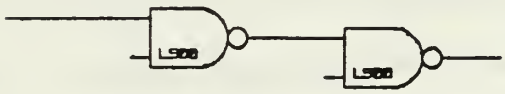
Signal names are used to place logical values (high or low) onto wires and are used to read logical values from a wire. Signal name conventions are:

Subscripts: Subscript ranges are enclosed in < >.

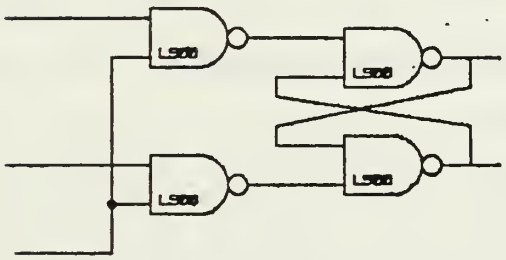
BUS<15..Ø>	16-bit signal
BUS<15>	MSB of BUS<15..Ø>
ST<3..Ø>	4-bit signal



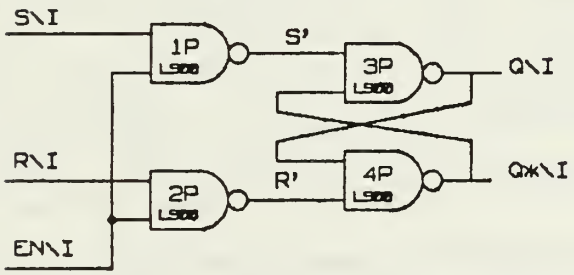
a) Positioning bodies



b) Attaching wires



c) Completed Drawing



d) Signal names attached. Drawing written

Figure A-1: SCALD System Circuit Diagram

Assertion: * indicates low assertion level.
SPC*
ENABLE*

Hierarchial: I is appended to signals that inter face hierarchial bodies to a low level circuit diagram.
G is appended to signals that are are global throughout the hierarchy
RESET I
CLOCK G

Once the drawing is completed, it is written. Writing a drawing simply stores the drawing onto the disk. Path properties are attached to each body by the SCALD system (1P, 2P, Figure B-1d) during the write process. The path properties identify each component and can be used to quickly find a particular component.

The designer can create a custom library by using hierarchial bodies. A low level circuit is drawn in GED (Figure A-1). The signal names that are to interface with the hierarchial body are appended with I. Unappended signals are invisible to a higher-level body. Figure A-2 shows a high-level hierarchial body for the circuit in Figure A-1. The hierarchial body can be used in another drawing and performs the same function as the circuit in the original drawing.

Any system to be simulated can be identified by a series of interconnected high-level bodies. The higher level bodies are defined by a series of interconnected low

level bodies. Hierarchial bodies allow complicated designs to be drawn in a manner easy to manipulate and understand.

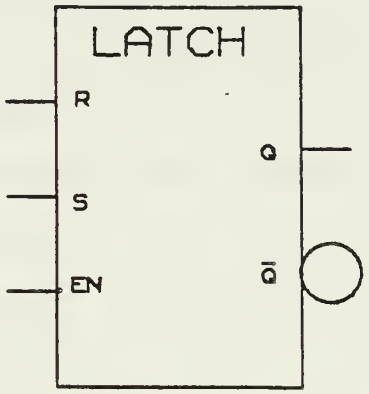


Figure A-2: SCALD Hierarchial Body

The simulator allows the designer to control the input signals and simulate the circuit for a certain length of time. The SCALD simulator accounts for the timing characteristics of each component and produces timing diagrams that can be used to verify the design or identify and correct errors. Timing diagrams are used throughout this thesis.

Time can be saved and multiple simulations can be quickly run using "script" files. A script file is a text file that provides a sequence of commands to the simulator. The key point is that simulations can be recreated or signals modified slightly and the effects on the outputs

can be studied with a minimum of effort. Without script files, the procedure shown above would have to be executed every time a signal is changed.

APPENDIX B: QUINE-MCCLUSKEY MINTERM REDUCTION ALGORITHM

Design of the voter turned out to be a seven variable problem, one variable too large for a standard Karnaugh map. A minterm reduction algorithm was therefore required. A check with several professors in the Electrical & Computer Engineering department and the Computer Science department failed to produce a copy of the algorithm. The NPS Computer Center performed a search with negative results.

The book by Fletcher, "An Engineering Approach to Digital Design" [Ref. 10] contained a boolean reduction algorithm. However, it was written in extended ALGOL 60 and could not be used.

This left the Quine-McCluskey tabulation method for minterm reduction. The book by Mano, "Digital Logic and Computer Design" [Ref. 4], explained the tabulation method very well and was used to write the algorithm in this appendix.

The Quine-McCluskey reduction algorithm is written in IBM basic and can be used with an Apple with minor changes to the algorithm. Unfortunately, the relatively small memory of the Apple II+ limits its use. The algorithm is very memory inefficient, but very interactive. Writing, testing, and using the algorithm turned out to be extremely time consuming. It was written as quickly as possible and

is very memory inefficient. No documentation for the algorithm was written.

The output files for MN1, MNØ, MX1, and MXØ are shown on pages 132 - 143. The next concern was to verify the correctness of the reductions. A "logic checker" algorithm was quickly written. It produces a list of minterms that is generated by MN1, MNØ, MX1 and MXØ. This list was then doublechecked against the original minterm list. The minterm list matched for all four functions. Although this proved that the functions were correct, it did not show if they are indeed minimum. A copy of the logic checker for MX1 is shown in pages 144 - 145 and the output of the checker is on pages 146 - 149.

```
1  PRINT"THIS  IS THE REDUCED SLICK VERSION USING  READ  AND
   DATA STATEMENTS"
2  PRINT"LOCATED IN STATEMENTS 205 AND 241-249":PRINT
3  PRINT"CHECK  THE DATA STATEMENTS TO ENSURE YOU  HAVE  THE
   DESIRED MINTERMS"
4  PRINT"ENSURE THAT THE LAST VALUE IN THE DATA STATEMENT IS
   NEGATIVE"
5  PRINT:INPUT"PRESS ANY KEY TO GO ON";UX$
10 REM  QUINE-MacCLUSKEY MINTERM REDUCTION ALGORITHM
20 REM  VIRGIL SPURLOCK, NAVAL POSTGRADUATE SCHOOL
30 REM  14 OCTOBER 1985, NO DOCUMENTATION WRITTEN
40 REM  DURING A RUN, IF THE ERROR "SUBSCRIPT OUT OF RANGE
   IN ____"
50 REM  APPEARS, INCREASE THE SIZE OF THE ARRAY IN LINES
   195 OR 200
60 REM  PROGRAM WILL RUN ON IBM BASIC
70 REM  PROGRAM WILL RUN ON APPLE II+ USING MBASIC BY MICRO
   SOFT (REQUIRES
80 REM  80 COLUMN CARD AND CPM (Z-80) CARD
90 CLS
100 CLEAR
110 INPUT"DO YOU WANT A HARDCOPY OF THIS TRANSACTION? (Y/N) "
    ;COPY$
120 IF COPY$ <> "Y" THEN GOTO 140
130 INPUT"DO YOU WANT THE LONG VERSION PRINTED (Y/N) ";XXXX$
140 REM
150 INPUT"HOW MANY BITS IN EACH MINTERM ";VA
160 NUMBER=2^VA
170 LARGE=NUMBER-1
180 DIM B(NUMBER)
190 PRINT:PRINT"INPUT THE MINTERMS, ONE AT A TIME.  INPUT A
    NEGATIVE NUMBER WHEN COMPLETED."
200 FOR I = 0 TO NUMBER
205 REM STATEMENT 210 HAS BEEN MODIFIED
210 READ B(I)
220 IF B(I) > LARGE THEN PRINT"MINTERM TOO LARGE.  REENTER":
    GOTO 210
230 IF B(I) < 0 THEN GOTO 250
240 NEXT I
241 REM
242 DATA 16,32,17,81,33,97,18,50,35,51,8,40,24,56,72,104,
    88,120,12,28,44,60,76,92,108,124,9,25,41,57,73,89,105,
    121,11,27,43,59,75,91,107,123,13,29,45,61,77,93,109,125,
    14,30,46,62,78,94,110,126,-9  :REM MX1
243 REM
244 DATA 16,64,17,81,18,50,66,98,67,83,4,68,20,84,36,100,52,
    116,12,28,44,60,76,92,108,124,6,22,38,54,70,86,102,118,
    7,23,39,55,71,87,103,119,13,29,45,61,77,93,109,125,14,
    30,46,62,78,94,110,126,-9  :REM MX0
245 REM
246 REM DATA 80,96,2,34,18,50,66,98,82,114,3,19,35,51,67,83,
```

```
    99,115,20,84,36,100,72,104,76,92,6,22,38,54,70,86,102,
    118,7,23,39,55,71,87,103,119,11,27,43,59,75,91,107,123,
    14,30,46,62,78,94,110,126,-9   :REM MN1
247 REM
248 DATA 48,96,1,65,17,81,33,97,49,113,3,19,35,51,67,83,99,
    115,36,100,24,56,72,104,44,60,7,23,39,55,71,87,103,119,
    9,25,41,57,73,89,105,121,11,27,43,59,75,91,107,123,13,
    29,45,61,77,93,109,125,-9  :REM MN0
249 REM
250 PRINT:PRINT"THERE ARE ";I;" TOTAL MINTERMS":PRINT
260 IF I > 2^VA THEN PRINT:PRINT:PRINT"****** ERROR ******"
270 IF I > 2^VA THEN PRINT:PRINT"TOO MANY MINTERMS FOR THE
    NUMBER OF BITS ENTERED.  RUN TERMINATED":END
280 PRINT "PRESS ANY KEY TO CONTINUE"
290 VIR$=INKEY$: IF VIR$="" THEN 290
294 REM
300 CPT=I-1:MAJ=0:PDQ=VA-1:VIRGIL=VA+1:PNT=0
310 DIM X$(2*I+I/2),Y$(VA),M$(VA,2*I),P$(2*I+I/2),Q$(VA,2*
    I),G$(VA,2*I),PI$(I)
320 DIM D(I,2*I),NONE(I+1),NO(I),GROUP(3*VA),SCAN(I)
330 CLS
340 GOSUB 4150
350 GOSUB 4340
360 PRINT
370 INPUT"REVIEW THE MINTERMS.  ARE THERE ANY ERRORS (Y/N)
    ";YUK$
380 IF YUK$="Y" THEN GOSUB 4460
390 IF YUK$<>"Y" THEN GOTO 440
400 INPUT"ARE THERE ANY MORE ERRORS ";MORE$
410 IF MORE$<>"Y" THEN GOTO 440
420 IF MORE$="Y" THEN GOSUB 4460
430 GOTO 400
440 GOSUB 4030
450 GOSUB 4150
460 PRINT"PRESS ANY KEY TO CONTINUE"
470 VIR$=INKEY$: IF VIR$="" THEN 470
480 CLS
485 PRINT "START TIME OF THE ROUTINE IS ";TIME$
486 IF COPY$="Y" THEN LPRINT:LPRINT TAB(10) "START TIME OF
    THE ROUTINE IS ";TIME$:LPRINT
490 PRINT:PRINT" THE MINTERMS ARE NEXT SORTED BY THE NUMBER
    OF ONES":PRINT
500 IF XXXX$="Y" THEN LPRINT:LPRINT:LPRINT"          THE
    MINTERMS ARE NEXT SORTED BY THE NUMBER OF ONES":LPRINT
510 CNT = -1
520 FOR J = 0 TO VA
530 COUNT=0
540 PRINT "--------------------"
550 IF XXXX$="Y" THEN LPRINT"          ====================="
560 FOR K = 0 TO I-1
570 IF NO(K)<>J THEN GOTO 660
```

```
580 CNT=CNT+1 : COUNT=COUNT+1
590 GROUP(J)=COUNT
600 D(J,COUNT)=B(K)
610 NONE(CNT)=NO(K)
620 M$(J,COUNT)=X$(K)
630 P$(CNT)=X$(K)
640 IF XXXX$="Y" THEN LPRINT TAB(10) M$(J,COUNT) ;"    "; D(
    J,COUNT)
650 PRINT M$(J,COUNT) ;"    ";D(J,COUNT)
660 NEXT K
670 NEXT J
680 IF XXXX$="Y" THEN LPRINT"          ==================="
690 FOR ABCDE = 1 TO VIRGIL
700 PRINT:PRINT"****  REDUCTION PASS NUMBER";ABCDE;"  ****"
710 IF XXXX$="Y" THEN LPRINT:LPRINT:LPRINT"          *****
    REDUCTION PASS NUMBER";ABCDE;" *****":LPRINT
720 IF XXXX$="Y" THEN LPRINT"          ====================="
730 AA=0:BB=0:CC=-1
740 GOSUB 1100
750 GOSUB 1750
760 VA=AA-1
770 FOR J=0 TO AA-1
780 GROUP(J)=GP(J)
790 NEXT J
800 FOR R = 0 TO AA-1
810 FOR S = 1 TO GP(R)
820 M$(R,S)=G$(R,S)
830 NEXT S
840 NEXT R
850 CPT=CC
860 FOR YEZ = 0 TO CC
870 X$(YEZ)=P$(YEZ)
880 NEXT YEZ
890 NEXT ABCDE
900 FOR K=1 TO I-1
910 SCAN(K)=0
920 FOR M=1 TO MAJ
930 D(M,K)=0
940 GROUP(M)=0
950 NEXT M
960 NEXT K
970 GOSUB 1910
980 GOSUB 2560
990 FOR BEER=0 TO I-1
1000 GOSUB 2900
1010 BEERMAN=0
1020 FOR TAP=1 TO MAJ
1030 FOR KEG=0 TO I-1
1040 BEERMAN=D(TAP,KEG) + BEERMAN
1050 NEXT KEG
1060 NEXT TAP
```

```
1070 IF BEERMAN=0 THEN GOSUB 3890
1080 NEXT BEER
1090 END
1100 REM ************************************************
1110 REM     SUBROUTINE PERFORMS MINTERM REDUCTION
1120 REM ************************************************
1130 FOR N1 = 0 TO VA-1
1140 CRZ = 0 : CRX = 0
1150 FOR J = 1 TO GROUP(N1)
1160 CRY = 0
1170 FOR K = 1 TO GROUP(N1+1)
1180 FLAG = 0:CRZ=CRZ+1:CXX=0
1190 Q$(N1,CRZ)=M$(N1,J)
1200 FOR L=1 TO PDQ+1
1210 IF MID$(M$(N1,J),L,1)<>MID$(M$(N1+1,K),L,1) THEN GOTO
     1230
1220 GOTO 1260
1230 MID$(Q$(N1,CRZ),L)="-"
1240 CXX=CXX+1
1250 IF CXX > 1 THEN FLAG=1
1260 NEXT L
1270 IF FLAG = 1 THEN CRZ = CRZ - 1:GOTO 1500
1280 BB=BB+1
1290 IF BB <= 0 THEN GOTO 1350
1300 FOR QQQ = 1 TO CRZ-1
1310 IF Q$(N1,CRZ)<>Q$(N1,QQQ) THEN GOTO 1340
1320 BB=BB-1
1330 GOTO 1490
1340 NEXT QQQ
1350 G$(AA,BB)=Q$(N1,CRZ)
1360 GP(AA)=BB
1370 IF XXXX$="Y" THEN LPRINT"          ";
1380 CC=CC+1
1390 REM
1400 P$(CC)=Q$(N1,CRZ)
1410 PRINT Q$(N1,CRZ);
1420 IF XXXX$="Y" THEN LPRINT Q$(N1,CRZ);
1430 IF ABCDE < 2 THEN GOTO 1470
1440 PRINT
1450 IF XXXX$="Y" THEN LPRINT
1460 GOTO 1490
1470 PRINT"   ";D(N1,J);",";D(N1+1,K)
1480 IF XXXX$="Y" THEN LPRINT "   ";D(N1,J);",";D(N1+1,K)
1490 GOSUB 1570
1500 NEXT K
1510 NEXT J
1520 PRINT"++++++++++++++++++++"
1530 IF XXXX$="Y" THEN LPRINT"          ====================="
1540 BB=0:AA=AA+1
1550 NEXT N1
1560 RETURN
```

```
1570 REM ************************************************
1580 REM    SUBROUTINE CREATES PRIME IMPLICANT LIST
1590 REM ************************************************
1600 HLD=0
1610 FOR RAY = 0 TO CPT
1620 IF X$(RAY)=M$(N1,J) THEN GOTO 1680
1630 NEXT RAY
1640 FOR RAY=0 TO CPT
1650 IF X$(RAY)=M$(N1+1,K) THEN GOTO  1680
1660 NEXT RAY
1670 GOTO 1730
1680 FOR MAY=RAY TO CPT-1
1690 X$(MAY)=X$(MAY+1)
1700 NEXT MAY
1710 CPT=CPT-1:HLD=HLD+1:IF HLD>=2 THEN GOTO 1730
1720 GOTO 1640
1730 RETURN
1740 REM ************************************************
1750 REM   SUBROUTINE THAT KEEPS MINTERMS
1760 REM ************************************************
1770 IF CPT=-1 THEN RETURN
1780 FOR YO=0 TO CPT
1790 MAJ=MAJ+1
1800 PRINT"PRIME IMPLICANT #";MAJ;"       ";
1810 IF XXXX$="Y" THEN LPRINT"          PI#";MAJ;"    ";
1820 PI$(MAJ)=X$(YO)
1830 PRINT PI$(MAJ);
1840 IF XXXX$="Y" THEN LPRINT PI$(MAJ);
1850 PRINT
1860 IF XXXX$="Y" THEN LPRINT
1870 NEXT YO
1880 RETURN
1890 IF XXXX$="Y" THEN LPRINT
1900 RETURN
1910 REM ************************************************
1920 REM  SUBROUTINE TO PRINT OUT THE PRIME IMPLICANTS
1930 REM ************************************************
1940 PRINT:PRINT"NOW PRINT THE LIST OF PRIME IMPLICANTS"
1950 IF XXXX$="Y" THEN LPRINT:LPRINT:LPRINT"          COMPL
     ETE LIST OF PRIME IMPLICANTS":LPRINT
1960 FOR T = 1 TO MAJ
1970 PRINT"PI #";T;"   ";
1980 IF XXXX$="Y" THEN LPRINT"          PI #";T;"   ";
1990 PRINT PI$(T);
2000 IF XXXX$="Y" THEN LPRINT PI$(T);
2010 PRINT"    ";
2020 IF XXXX$="Y" THEN LPRINT"    ";
2030 GOSUB 2080
2040 PRINT
2050 IF XXXX$="Y" THEN LPRINT
2060 NEXT T
```

```
2070 RETURN
2080 REM ************************************************
2090 REM     SUBROUTINE COMPUTES DECIMAL EQUIVALENTS
2100 REM ************************************************
2110 SUM=0
2120 SD=0  :REM  THIS COUNTS THE NUMBER OF DASHES
2130 FOR J=1 TO PDQ+1
2140 IF MID$(PI$(T),J,1)="-" THEN SD=SD+1
2150 NEXT J
2160 IF SD=0 THEN TMT=1
2170 IF SD>0 THEN TMT=2^SD
2180 FOR X=0 TO PDQ
2190 Y=ABS(X-PDQ)
2200 IF MID$(PI$(T),X+1,1)="0" THEN GOTO 2240
2210 IF MID$(PI$(T),X+1,1)="1" THEN GOTO 2240
2220 REM HERE IS WHERE THE DASHES ARE HANDLED
2230 SD=SD-1:GOTO 2280
2240 FOR J=1 TO TMT
2250 MID$(X$(J),X+1,1)=MID$(PI$(T),X+1,1)
2260 NEXT J
2270 GOTO 2390
2280 CNT=0
2290 TOGGLE$="0"
2300 FOR J=1 TO TMT
2310 MID$(X$(J),X+1,1)=TOGGLE$
2320 CNT=CNT+1
2330 IF CNT >= 2^SD THEN GOTO 2350
2340 GOTO 2380
2350 IF TOGGLE$="1" THEN TOGGLE$="0": GOTO 2370
2360 IF TOGGLE$="0" THEN TOGGLE$="1"
2370 CNT=0
2380 NEXT J
2390 NEXT X
2400 FOR M=1 TO TMT
2410 SUM=0
2420 FOR J=0 TO PDQ
2430 X=ABS(J-PDQ)
2440 F=ASC(MID$(X$(M),J+1,1))-48
2450 SUM=SUM+F*(2^X)
2460 NEXT J
2470 NONE(M)=SUM
2480 NEXT M
2490 FOR J=1 TO TMT
2500 D(T,J)=NONE(J)
2510 NO(T)=TMT
2520 PRINT NONE(J) ",";
2530 IF XXXX$="Y" THEN LPRINT NONE(J) ",";
2540 NEXT J
2550 RETURN
2560 REM ************************************************
2570 REM     SUBROUTINE DEVELOPS PRIME IMPLICANT TABLE
```

```
2580 REM ************************************************
2590 PRINT:PRINT:PRINT:PRINT:
2600 IF COPY$="Y" THEN LPRINT:LPRINT:LPRINT TAB(10) "PRIME
     IMPLICANT TABLE":LPRINT
2610 FOR J=0 TO I-1
2620 PRINT TAB(14+J*4) B(J);
2630 IF COPY$="Y" THEN LPRINT TAB(14+J*4) B(J);
2640 NEXT J
2650 FOR J=0 TO I-1
2660 PRINT TAB(14+J*4) "----";
2670 IF COPY$="Y" THEN LPRINT TAB(14+J*4) "----";
2680 NEXT J
2690 FOR J=1 TO MAJ
2700 PRINT TAB(10) J;
2710 IF COPY$="Y" THEN LPRINT TAB(10) J;
2720 FOR R=1 TO NO(J)
2730 FOR S=0 TO I-1
2740 IF D(J,R)=B(S) THEN SCAN(S)=1: PRINT TAB(14+S*4) " X";
2750 IF D(J,R)=B(S) THEN IF COPY$="Y" THEN SCAN(S)=1: LPRINT
     TAB(14+S*4) " X";
2760 NEXT S
2770 NEXT R
2780 PRINT
2790 FOR H=0 TO I-1
2800 D(J,H)=0
2810 NEXT H
2820 FOR H=0 TO I-1
2830 IF SCAN(H)=1 THEN D(J,H)=1
2840 SCAN(H)=0
2850 NEXT H
2860 IF COPY$="Y" THEN LPRINT
2870 NEXT J
2880 IF COPY$="Y" THEN LPRINT:LPRINT
2890 RETURN
2900 REM ************************************************
2910 REM   SUBROUTINE PERFORMS REDUCTION ON PI TABLE
2920 REM ************************************************
2930 FOR Z=0 TO I-1
2940 SCAN(Z)=0
2950 NEXT Z
2960 GOTO 3070
2970 IF XXXX$="Y" THEN LPRINT:LPRINT
2980 FOR R=1 TO MAJ
2990 IF XXXX$="Y" THEN LPRINT TAB(12) " ";
3000 FOR S=0 TO I-1
3010 PRINT D(R,S) "  ";
3020 IF XXXX$="Y" THEN LPRINT D(R,S) " ";
3030 NEXT S
3040 PRINT
3050 IF XXXX$="Y" THEN LPRINT
3060 NEXT R
```

```
3070 FOR J=0 TO I-1
3080 SUM=0
3090 FOR K=1 TO MAJ
3100 SUM = D(K,J) + SUM
3110 NEXT K
3120 B(J)=SUM       :REM PRINT "J=";J;"  B(J)=;B(J)
3130 NEXT J
3140 REM FIRST PASS REDUCTION ON PI TABLE
3150 XCHK=0
3160 FOR XRAY=1 TO I-1
3170 FOR M=1 TO MAJ
3180 FOR J=0 TO I-1
3190 IF B(J)<>XRAY THEN GOTO 3290
3200 XCHK=XCHK+1
3210 MING=0
3220 IF XRAY>1 THEN GOSUB 3490
3230 IF MING=5 THEN RETURN
3240 FOR K=1 TO MAJ
3250 IF D(K,J)<>1 THEN GOTO 3280
3260 GROUP(K)=1
3270 GOTO 3290
3280 NEXT K
3290 NEXT J
3300 IF XCHK=0 THEN GOTO 3470
3310 NEXT M
3320 PRINT
3330 FOR J=1 TO MAJ
3340 IF GROUP(J) <> 1 THEN GOTO 3390
3350 FOR K=0 TO I-1
3360 IF D(J,K)=1 THEN SCAN(K)=1:D(J,K)=0
3370 D(J,K)=0
3380 NEXT K
3390 NEXT J
3400 FOR K=0 TO I-1
3410 IF SCAN(K)<>1 THEN GOTO 3450
3420 FOR J=1 TO MAJ
3430 D(J,K)=0
3440 NEXT J
3450 NEXT K
3460 IF XCHK>0 THEN GOTO 3480
3470 NEXT XRAY
3480 RETURN
3490 REM ************************************************
3500 REM    SUBROUTINE HANDLES MULTIPLE COLUMN CHOICES
3510 REM ************************************************
3520 FOR C=0 TO I-1
3530 NO(C)=0
3540 NONE(C)=0
3550 NEXT C
3560 FOR M=1 TO MAJ
3570 FOR J=0 TO I-1
```

```
3580 IF B(J)<>XRAY THEN GOTO 3630
3590 FOR K=1 TO MAJ
3600 IF D(K,J)<>1 THEN GOTO 3620
3610 NO(K)=1
3620 NEXT K
3630 NEXT J
3640 NEXT M
3650 FOR E=1 TO MAJ
3660 SUM=0
3670 IF NO(E) <> 1 THEN GOTO 3720
3680 FOR F=0 TO I-1
3690 SUM=D(E,F)+SUM
3700 NEXT F
3710 NONE(E)=SUM
3720 NEXT E
3730 MAX=0
3740 FOR G=1 TO MAJ
3750 IF NONE(G) >= MAX THEN MAX=NONE(G):HOLD=G
3760 NEXT G
3770 GROUP(HOLD)=1
3780 FOR A=0 TO I-1
3790 IF D(HOLD,A)=1 THEN SCAN(A)=1:D(HOLD,A)=0
3800 NEXT A
3810 FOR B=0 TO I-1
3820 IF SCAN(B)<>1 THEN GOTO 3860
3830 FOR C=1 TO MAJ
3840 D(C,B)=0
3850 NEXT C
3860 NEXT B
3870 MING=5
3880 RETURN
3890 REM ************************************************
3900 REM   SUBROUTINE PRINTS THE FINAL PRODUCT
3910 REM ************************************************
3920 PRINT:PRINT:PRINT "***** FINAL IMPLICANT LIST *****":PRINT
3930 IF COPY$="Y" THEN LPRINT TAB(10) "***** FINAL IMPLICANT
     LIST *****":LPRINT
3940 FOR A=1 TO MAJ
3945 IF GROUP(A) <> 1 THEN GOTO 4000
3950 PRINT TAB(10) "    ";PI$(A)
3960 IF COPY$="Y" THEN LPRINT TAB(15) PI$(A)
4000 NEXT A
4002 PRINT "RUN COMPLETED AT ";TIME$
4004 IF COPY$="Y" THEN LPRINT:LPRINT TAB(10) "RUN COMPLETED
     AT ";TIME$:LPRINT
4010 END
4020 RETURN
4030 REM ************************************************
4040 REM           SUBROUTINE  BUBBLE SORT
4050 REM ************************************************
4060 FOR Y=0 TO I-2
```

```
4070 FOR X=Y+1 TO I-1
4080 IF B(Y) < B(X) THEN GOTO 4120
4090 TEMP=B(Y):TEM$=X$(Y)
4100 B(Y)=B(X):X$(Y)=X$(X)
4110 B(X)=TEMP:X$(X)=TEMP$
4120 NEXT X
4130 NEXT Y
4140 RETURN
4150 REM ******************************************
4160 REM   SUBROUTINE  CREATES BINARY EQUIVALENTS
4170 REM ******************************************
4180 Y$(VA)=""
4190 FOR J=0 TO I-1
4200 NUMONE=0:CNT=-1:BTEMP=B(J)
4210 FOR M= VA-1 TO 0 STEP -1
4220 CNT=CNT+1
4230 IF BTEMP - 2^M <0 THEN GOTO 4280
4240 Y$(M)=Y$(M+1)+"1"
4250 BTEMP=BTEMP-2^M
4260 NUMONE=NUMONE+1
4270 GOTO 4290
4280 Y$(M)=Y$(M+1)+"0"
4290 NEXT M
4300 X$(J)=Y$(M+1)
4310 NO(J)=NUMONE
4320 NEXT J
4330 RETURN
4340 REM ******************************************
4350 REM   SUBROUTINE PRINTS OUT MINTERMS
4360 REM ******************************************
4370 IF PNT=1 THEN PRINT"CORRECTED LIST OF MINTERMS":GOTO 4390
4380 PRINT"LIST OF MINTERMS"
4390 IF PNT=1 THEN IF COPY$="Y" THEN LPRINT:LPRINT TAB(10)
     "CORRECTED LIST OF MINTERMS ":LPRINT: GOTO 4410
4400 IF COPY$="Y" THEN LPRINT:LPRINT TAB(10) "LIST OF MIN
     TERMS":LPRINT
4410 FOR J=0 TO I-1
4420 PRINT "(";J+1;")" TAB(10) X$(J) TAB(14+PDQ) B(J)
4430 IF COPY$="Y" THEN LPRINT TAB(10) "(";J+1;")" TAB(20)
     X$(J) TAB(24+PDQ) B(J)
4440 NEXT J
4450 RETURN
4460 REM ******************************************
4470 REM    SUBROUTINE IF ERROR IN MINTERM INPUT
4480 REM ******************************************
4490 PNT=1
4500 PRINT"TERMINATE THE CORRECTION SESSION BY ENTERING A
     NEGATIVE NUMBER"
4510 INPUT"ENTER THE NUMBER OF THE INCORRECT MINTERM ";YECCH
4520 IF YECCH < 0 THEN GOTO 4570
4530 INPUT"ENTER THE CORRECT MINTERM VALUE ";VALUE
```

```
4540 B(YECCH-1)=VALUE
4550 IF VALUE < 0 THEN GOTO 4570
4560 GOTO 4510
4570 GOSUB 4340
4580 RETURN
```

Output File for MN1

REDUCTION OF STATE1 VARIABLES
OUTPUT MN1 25 OCT 85

LIST OF MINTERMS

(1)	1010000	80
(2)	1100000	96
(3)	0000010	2
(4)	0100010	34
(5)	0010010	18
(6)	0110010	50
(7)	1000010	66
(8)	1100010	98
(9)	1010010	82
(10)	1110010	114
(11)	0000011	3
(12)	0010011	19
(13)	0100011	35
(14)	0110011	51
(15)	1000011	67
(16)	1010011	83
(17)	1100011	99
(18)	1110011	115
(19)	0010100	20
(20)	1010100	84
(21)	0100100	36
(22)	1100100	100
(23)	1001000	72
(24)	1101000	104
(25)	1001100	76
(26)	1011100	92
(27)	0000110	6
(28)	0010110	22
(29)	0100110	38
(30)	0110110	5$
(31)	1000110	70
(32)	1010110	86
(33)	1100110	102
(34)	1110110	118
(35)	0000111	7
(36)	0010111	23
(37)	0100111	39
(38)	0110111	55

```
( 39 )    1000111    71
( 40 )    1010111    87
( 41 )    1100111    103
  42 )    1110111    119
( 43 )    0001011    11
( 44 )    0011011    27
( 45 )    0101011    43
( 46 )
( 47 )    1001011    75
( 48 )    1011011    91
( 49 )    1101011    107
( 50 )    1111011    123
( 51 )    0001110    14
( 52 )    0011110    30
( 53 )    0101110    46
( 54 )    0111110    62
( 55 )    1001110    78
( 56 )    1011110    94
( 57 )    1101110    110
( 58 )    1111110    126
```

START TIME OF THE ROUTINE IS 10:51:25

```
PI#  1    1001-00     72 , 76
PI#  2    1-01000     72 , 104
PI#  3    110-000     96 , 104
PI#  4    -0101-0     20 , 22 , 84 , 86
PI#  5    -1001-0     36 , 38 , 100 , 102
PI#  6    1010--0     80 , 82 , 84 , 86
PI#  7    1100--0     96 , 98 , 100 , 102
PI#  8    10-11-0     76 , 78 , 92 , 94
PI#  9    101-1-0     84 , 86 , 92 , 94
PI#  10   ----011     3 , 11 , 19 , 27 , 35 , 43 , 51 , 59
                      67 , 75 , 83 , 91 , 99 , 107 , 115 , 123
PI#  11   ----110     6 , 14 , 22 , 30 , 38 , 46 , 54 , 62
                      70 , 78 , 86 , 94 , 102 , 110 , 118 , 126
PI#  12   ---0-1-     2 , 3 , 6 , 7 , 18 , 19 , 22 , 23 , 34
                      35 , 38 , 39 , 50 , 51 , 54 , 55 , 66
                      67 , 70 , 71 , 82 , 83 , 86 , 87 , 98
                      99 , 102 , 103 , 114 , 115 , 118 , 119
```

```
***** FINAL IMPLICANT LIST *****

     1-01000
     110-000
     -0101-0
     -1001-0
     1010--0
     10-11-0
     ----011
     ----110
     ---0-1-

RUN COMPLETED AT 12:10:48
```

OUTPUT File for MNØ

REDUCTION OF STATEl VARIABLES
OUTPUT MNØ

LIST OF MINTERMS

(1)	0110000	48
(2)	1100000	96
(3)	0000001	1
(4)	1000001	65
(5)	0010001	17
(6)	1010001	81
(7)	0100001	33
(8)	1100001	97
(9)	0110001	49
(10)	1110001	113
(11)	0000011	3
(12)	0010011	19
(13)	0100011	35
(14)	0110011	51
(15)	1000011	67
(16)	1010011	83
(17)	1100011	99
(18)	1110011	115
(19)	0100100	36
(20)	1100100	100
(21)	0011000	24
(22)	0111000	56
(23)	1001000	72
(24)	1101000	104
(25)	0101100	44
(26)	0111100	60
(27)	0000111	7
(28)	0010111	23
(29)	0100111	39
(30)	0110111	55
(31)	1000111	71
(32)	1010111	87
(33)	1100111	103
(34)	1110111	119
(35)	0001001	9
(36)	0011001	25
(37)	0101001	41

```
( 38 )     0111001     57
( 39 )     1001001     73
( 40 )     1011001     89
( 41 )     1101001     105
( 42 )     1111001     121
( 43 )     0001011     11
( 44 )     0011011     27
( 45 )     0101011     43
( 46 )     0111011     59
( 47 )     1001011     75
( 48 )     1011011     91
( 49 )     1101011     107
( 50 )     1111011     123
( 51 )     0001101     13
( 52 )     0011101     29
( 53 )     0101101     45
( 54 )     0111101     61
( 55 )     1001101     77
( 56 )     1011101     93
( 57 )     1101101     109
( 58 )     1111101     125
```

START TIME OF THE ROUTINE IS 20:23:11

```
PI#  1     010-100     36 , 44
PI#  2     -100100     36 , 100
PI#  3     1100-00     96 , 100
PI#  4     0-1100-     24 , 25 , 56 , 57
PI#  5     011-00-     48 , 49 , 56 , 57
PI#  6     1-0100-     72 , 73 , 104 , 105
PI#  7     110-00-     96 , 97 , 104 , 105
PI#  8     01-110-     44 , 45 , 60 , 61
PI#  9     0111-0-     56 , 57 , 60 , 61
PI#  10    ---0-11     3 , 7 , 19 , 23 , 35 , 39 , 51 , 55
                       67 , 71 , 83 , 87 , 99 , 103 , 115 , 119
PI#  11    ---1-01     9 , 13 , 25 , 29 , 41 , 45 , 57 , 61
                       73 , 77 , 89 , 93 , 105 , 109 , 121 , 125
PI#  12    ----0-1     1 , 3 , 9 , 11 , 17 , 19 , 25 , 27 , 33
                       35 , 41 , 43 , 49 , 51 , 57 , 59 , 65
                       67 , 73 , 75 , 81 , 83 , 89 , 91 , 97
                       99 , 105 , 107 , 113 , 115 , 121 , 123
```

```
***** FINAL IMPLICANT LIST *****

     -100100
     1100-00
     0-1100-
     011-00-
     1-0100-
     01-110-
     ---0-11
     ---1-01
     ----0-1

RUN COMPLETED AT 21:42:32
```

Output File for MX1

REDUCTION OF STATE1 VARIABLES.
OUTPUT MX1 25 OCT 85

LIST OF MINTERMS

(1)	0010000	16
(2)	0100000	32
(3)	0010001	17
(4)	1010001	81
(5)	0100001	33
(6)	1100001	97
(7)	0010010	18
(8)	0110010	50
(9)	0100011	35
(10)	0110011	51
(11)	0001000	8
(12)	0101000	40
(13)	0011000	24
(14)	0111000	56
(15)	1001000	72
(16)	1101000	104
(17)	1011000	88
(18)	1111000	120
(19)	0001100	12
(20)	0011100	28
(21)	0101100	44
(22)	0111100	60
(23)	1001100	76
(24)	1011100	92
(25)	1101100	108
(26)	1111100	124
(27)	0001001	9
(28)	0011001	25
(29)	0101001	41
(30)	0111001	57
(31)	1001001	73
(32)	1011001	89
(33)	1101001	105
(34)	1111001	121
(35)	0001011	11

```
( 36 )     00110110     27
( 37 )     01010110     43
( 38 )     01110110     59
( 39 )     10010110     75
( 40 )     10110110     91
( 41 )     11010110     107
( 42 )     11110110     123
( 43 )     00011010     13
( 44 )     00111010     29
( 45 )     01011010     45
( 46 )     01111010     61
( 47 )     10011010     77
( 48 )     10111010     93
( 49 )     11011010     109
( 50 )     11111010     125
( 51 )     00011100     14
( 52 )     00111100     30
( 53 )     01011100     46
( 54 )     01111100     62
( 55 )     10011100     78
( 56 )     10111100     94
( 57 )     11011100     110
( 58 )     11111100     126

START TIME OF THE ROUTINE IS 16:04:11

PI#  1     00100-0     16 , 18
PI#  2     0-10010     18 , 50
PI#  3     01100 1-    50 , 51
PI#  4     001-00-     16 , 17 , 24 , 25
PI#  5     010-00-     32 , 33 , 40 , 41
PI#  6     -01-001     17 , 25 , 81 , 89
PI#  7     010-0-1     33 , 35 , 41 , 43
PI#  8     -10-001     33 , 41 , 97 , 105
PI#  9     01--011     35 , 43 , 51 , 59
PI#  10    ---10-1     9 , 11 , 25 , 27 , 41 , 43 ,
                       57 , 59 ,73 , 75 , 89 , 91 ,
                       105 , 107 ,121 , 123 ,
PI#  11    ---11-0     12 , 14 , 28 , 30 , 44 , 46 ,
                       60 , 62 , 76 , 78 , 92 , 94 ,
                       108 , 110 , 124 , 126
PI#  12    ---1-0-     8 , 9 , 12 , 13 , 24 , 25 ,
                       28 , 29 , 40 , 41 , 44 , 45 ,
                       56 , 57 , 60 , 61 , 72 , 73 ,
                       76 , 77 , 88 , 89 , 92 , 93 ,
                       104 , 105 , 108 , 109 , 120 ,
                       121 , 124 , 125
```

***** FINAL IMPLICANT LIST *****

```
0-10010
001-00-
010-00-
-01-001
-10-001
01--011
---10-1
---11-0
---1-0-
```

RUN COMPLETED AT 17:21:03

Output for MX0

REDUCTION OF STATE1 VARIABLES
OUTPUT MX0

LIST OF MINTERMS

(1)	0010000	16
(2)	1000000	64
(3)	0010001	17
(4)	1010001	81
(5)	0010010	18
(6)	0110010	50
(7)	1000010	66
(8)	1100010	98
(9)	1000011	67
(10)	1010011	83
(11)	0000100	4
(12)	1000100	68
(13)	0010100	20
(14)	1010100	84
(15)	0100100	36
(16)	1100100	100
(17)	0110100	52
(18)	1110100	116
(19)	0001100	12
(20)	0011100	28
(21)	0101100	44
(22)	0111100	60
(23)	1001100	76
(24)	1011100	92
(25)	1101100	108
(26)	1111100	124
(27)	0000110	6
(28)	0010110	22
(29)	0100110	38
(30)	0110110	54
(31)	1000110	70
(32)	1010110	86
(33)	1100110	102
(34)	1110110	118
(35)	0000111	7
(36)	0010111	23

```
( 37 )     0100111     39
( 38 )     0110111     55
( 39 )     1000111     71
( 40 )     1010111     87
( 41 )     1100111     103
( 42 )     1110111     119
( 43 )     0001101     13
( 44 )     0011101     29
( 45 )     0101101     45
( 46 )     0111101     61
( 47 )     1001101     77
( 48 )     1011101     93
( 49 )     1101101     109
( 50 )     1111101     125
( 51 )     0001110     14
( 52 )     0011110     30
( 53 )     0101110     46
( 54 )     0111110     62
( 55 )     1001110     78
( 56 )     1011110     94
( 57 )     1101110     110
( 58 )     1111110     126

START TIME OF THE ROUTINE IS 18:27:51

PI#  1          001000-     16 , 17
PI#  2          -010001     17 , 81
PI#  3          10100-1     81 , 83
PI#  4          0010--0     16 , 18 , 20 , 22
PI#  5          1000--0     64 , 66 , 68 , 70
PI#  6          0-10-10     18 , 22 , 50 , 54
PI#  7          1000-1-     66 , 67 , 70 , 71
PI#  8          1-00-10     66 , 70 , 98 , 102
PI#  9          10-0-11     67 , 71 , 83 , 87
PI#  10         ---011-     6 , 7 , 22 , 23 , 38 , 39 , 54 ,
                            55 , 70 , 71 , 86 , 87 , 102 ,
                            103 , 118 , 119 ,
PI#  11         ---110-     12 , 13 , 28 , 29 , 44 , 45 , 60 ,
                            61 , 76 , 77 , 92 , 93 , 108 ,
                            109 , 124 , 125 ,
PI#  12         ----1-0     4 , 6 , 12 , 14 , 20 , 22 , 28 ,
                            30 , 36 , 38 , 44 , 46 , 52 , 54 ,
                            60 , 62 , 68 , 70 , 76 , 78 , 84 ,
                            86 , 92 , 94 , 100 , 102 , 108 ,
                            110 , 116 , 118 , 124 , 126
```

***** FINAL IMPLICANT LIST *****

-010001
0010--0
1000--0
0-10-10
1-00-10
10-0-11
---011-
---110-
----1-0

RUN COMPLETED AT 19:45:33

LOGIC CHECKER ALGORITHM USING MX1

```
5 OPEN "LOGICCHK.TXT" FOR OUTPUT AS #1
6 PRINT #1,"PERFORMS LOGIC CHECK FOR STATE1 VARIABLES"
7 PRINT #1,"MX1"
10 VA=7
20 I=2^VA
30 DIM B(I),X$(I),Y$(VA),NO(I)
40 FOR K=0 TO I-1
50 B(K)=K
60 NEXT K
70 REM ******************************************
80 REM   SUBROUTINE  CREATES BINARY EQUIVALENTS
90 REM ******************************************
100 Y$(VA)=""
110 FOR J=0 TO I-1
120 NUMONE=0:CNT=-1:BTEMP=B(J)
130 FOR M= VA-1 TO 0 STEP -1
140 CNT=CNT+1
150 IF BTEMP - 2^M <0 THEN GOTO 200
160 Y$(M)=Y$(M+1)+"1"
170 BTEMP=BTEMP-2^M
180 NUMONE=NUMONE+1
190 GOTO 210
200 Y$(M)=Y$(M+1)+"0"
210 NEXT M
220 X$(J)=Y$(M+1)
230 PRINT X$(J) TAB(4+VA) J TAB(9+VA);
235 PRINT #1, X$(J) TAB(4+VA) J TAB(9+VA);
240 NO(J)=NUMONE
245 GOSUB 500
250 NEXT J
260 CLOSE #1
270 END
500 REM ***** SUBROUTINE THAT PRODUCES LOGICAL VARIABLES *****
510 FOR K=1 TO VA
520 HOLD$=MID$(X$(J),K,1)
525 I(K)=ASC(HOLD$)-48
530 NEXT K
600 A=I(1)
610 B=I(2)
620 C=I(3)
630 D=I(4)
640 E=I(5)
650 F=I(6)
660 G=I(7)
800 REM ******* COMPLETES THE LOGICAL OPERATORS ********
810 IF A=0 THEN ABAR=1
820 IF A=1 THEN ABAR=0
830 IF B=0 THEN BBAR=1
```

```
840 IF B=1 THEN BBAR=0
850 IF C=0 THEN CBAR=1
860 IF C=1 THEN CBAR=0
870 IF D=0 THEN DBAR=1
880 IF D=1 THEN DBAR=0
890 IF E=0 THEN EBAR=1
900 IF E=1 THEN EBAR=0
910 IF F=0 THEN FBAR=1
920 IF F=1 THEN FBAR=0
930 IF G=1 THEN GBAR=0
940 IF G=0 THEN GBAR=1
1000 REM **** ACTUALLY COMPUTES THE LOGICAL FUNCTION *****
1010 AND1=ABAR*C*DBAR*EBAR*F*GBAR
1020 AND2=ABAR*BBAR*C*EBAR*FBAR
1030 AND3=ABAR*B*CBAR*EBAR*FBAR
1040 AND4=BBAR*C*EBAR*FBAR*G
1050 AND5=B*CBAR*EBAR*FBAR*G
1060 AND6=ABAR*B*EBAR*F*G
1070 AND7=D*EBAR*G
1080 AND8=D*E*GBAR
1090 AND9=D*FBAR
1100 OUTPUT=AND1+AND2+AND3+AND4+AND5+AND6+AND7+AND8+AND9
1199 IF OUTPUT>1 THEN OUTPUT=1
1200 REM **** PRINTS THE OUTPUT ****
1210 PRINT " ";OUTPUT
1220 PRINT #1," ";OUTPUT
1999 RETURN
```

OUTPUT FILE FOR LOGICAL CHECKER
FOR STATE1, MX1

0	0000000	0
1	0000001	0
2	0000010	0
3	0000011	0
4	0000100	0
5	0000101	0
6	0000110	0
7	0000111	0
8	0001000	1
9	0001001	1
10	0001010	0
11	0001011	1
12	0001100	1
13	0001101	1
14	0001110	1
15	0001111	0
16	0010000	1
17	0010001	1
18	0010010	1
19	0010011	0
20	0010100	0
21	0010101	0
22	0010110	0
23	0010111	0
24	0011000	1
25	0011001	1
26	0011010	0
27	0011011	1
28	0011100	1
29	0011101	1
30	0011110	1
31	0011111	0
32	0100000	1
33	0100001	1
34	0100010	0
35	0100011	1
36	0100100	0
37	0100101	0
38	0100110	0
39	0100111	0
40	0101000	1
41	0101001	1
42	0101010	0
43	0101011	1

44	0101100	1
45	0101101	1
46	0101110	1
47	0101111	0
48	0110000	0
49	0110001	0
50	0110010	1
51	0110011	0
52	0110100	0
53	0110101	0
54	0110110	0
55	0110111	0
56	0111000	1
57	0111001	1
58	0111010	0
59	0111011	1
60	0111100	1
61	0111101	1
62	0111110	1
63	0111111	0
64	1000000	0
65	1000001	0
66	1000010	0
67	1000011	0
68	1000100	0
69	1000101	0
70	1000110	0
71	1000111	0
72	1001000	1
73	1001001	1
74	1001010	0
75	1001011	1
76	1001100	1
77	1001101	1
78	1001110	1
79	1001111	0
80	1010000	0
81	1010001	1
82	1010010	0
83	1010011	0
84	1010100	0
85	1010101	0
86	1010110	0
87	1010111	0
88	1011000	1
89	1011001	1
90	1011010	0
91	1011011	1
92	1011100	1
93	1011101	1
94	1011110	1
95	1011111	0

96	1100000	0
97	1100001	1
98	1100010	0
99	1100011	0
100	1100100	0
101	1100101	0
102	1100110	0
103	1100111	0
104	1101000	1
105	1101001	1
106	1101010	0
108	1101100	1
109	1101101	1
110	1101110	1
111	1101111	0
112	1110000	0
113	1110001	0
114	1110010	0
115	1110011	0
116	1110100	0
117	1110101	0
118	1110110	0
119	1110111	0
120	1111000	1
121	1111001	1
122	1111010	0
123	1111011	1
124	1111100	1
125	1111101	1
126	1111110	1
127	1111111	0

APPENDIX C: PROM MEMORY CONTENTS

The Scald logic simulator loads memories from a Unix file. Each file begins with a line that gives the bit range for the memory cells of the PROM and is followed by one or more memory block definitions. The "MEM BLOCK a,b" command starts a memory cell by defining a block of memory with consecutive addresses. The letter "a" is the starting address in decimal and "b" is the number of addresses in the block, again in decimal. The "END MEM BLOCK" command defines the end of the memory block. Memory blocks are used continuously throughout the file.

The first PROM (27S291.89p) is loaded with the contents of the unix file prom1.dat. PROM 27S291.88p is loaded from prom2.dat, PROM 27S291.87p is loaded from prom3.dat and PROM 27S291.86p is loaded from prom4.dat. The four Unix files are listed on the following twelve pages as shown below:

```
FILE_TYPE = MEMORY_CONTENTS;
BIT_RANGE = 7..0;

MEM_BLOCK 0,1;
  0111 1010;
END_MEM_BLOCK;

MEM_BLOCK 512,16;
  0111 1010;
  0111 1010;
  0111 1010;
  0111 1010;
  0111 1010;
  0111 1010;
  0111 1010;
  0111 1010;
  0111 1010;
  0111 1010;
  0111 1010;
  0111 1010;
  0111 1010;
  0111 1010;
  0111 1010;
  0111 1010;
END_MEM_BLOCK;

MEM_BLOCK 780,1;
  0011 1010;
END_MEM_BLOCK;

MEM_BLOCK 1024,1;
  0111 1010;
END_MEM_BLOCK;

MEM_BLOCK 1040,1;
  0111 1010;
END_MEM_BLOCK;

MEM_BLOCK 1048,1;
  0111 1010;
END_MEM_BLOCK;

MEM_BLOCK 1056,1;
  0111 1010;
END_MEM_BLOCK;

MEM_BLOCK 1072,1;
  0111 1010;
END_MEM_BLOCK;

MEM_BLOCK 1088,1;
  0111 1010;
END_MEM_BLOCK;

MEM_BLOCK 1104,1;
  0111 1010;
END_MEM_BLOCK;

MEM_BLOCK 1120,1;
  0111 1010;
END_MEM_BLOCK;

MEM_BLOCK 1136,1;
  0111 1010;
```

```
END_MEM_BLOCK;

MEM_BLOCK 1536,3;
  Ø111 1ØØØ;
  Ø111 1ØØØ;
  Ø111 1Ø1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1568,8;
  Ø111 1Ø1Ø;
  Ø111 1Ø1Ø;
  Ø111 1Ø1Ø;
  Ø111 1Ø1Ø;
  Ø111 1Ø1Ø;
  Ø111 1Ø1Ø;
  Ø111 1Ø1Ø;
  Ø111 1Ø1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1584,4;
  ØlØ1 1Ø1Ø;
  Ø11Ø 1Ø1Ø;
  Ø111 1Ø1Ø;
  1111 1Ø11;
END_MEM_BLOCK;

MEM_BLOCK 16ØØ,15;
  Ø111 1Ø1Ø;
  Ø111 1Ø1Ø;
  Ø111 1Ø1Ø;
  Ø111 1Ø1Ø;
  Ø111 1Ø1Ø;
  Ø111 1Ø1Ø;
  Ø111 1Ø1Ø;
  Ø111 1Ø1Ø;
  ØØ11 1Ø1Ø;
  ØØ11 1Ø1Ø;
  Ø111 1ØØØ;
  Ø111 1ØØØ;
  Ø111 1ØØØ;
  Ø111 1ØØØ;
  Ø111 1Ø1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1616,5;
  Ø111 1Ø1Ø;
  Ø111 1Ø1Ø;
  Ø111 1Ø1Ø;
  Ø111 1Ø11;
  Ø111 1Ø1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1632,3;
  Ø111 1Ø1Ø;
  Ø111 1Ø11;
  Ø111 1Ø1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1648,3;
  Ø111 1Ø1Ø;
  Ø111 1Ø11;
  Ø111 1Ø1Ø;
END_MEM_BLOCK;
```

```
MEM_BLOCK 1843,1Ø;
   1111 1Ø1Ø;
   1111 1Ø1Ø;
   1Ø11 1Ø1Ø;
   1Ø11 1Ø1Ø;
   1111 1ØØØ;
   1111 1ØØØ;
   1111 1ØØØ;
   1111 1ØØØ;
   1Ø11 1Ø1Ø;
   ØØ11 1Ø1Ø;
END_MEM_BLOCK;

END.
```

```
FILE_TYPE = MEMORY_CONTENTS;
BIT_RANGE = 7..Ø;

MEM_BLOCK Ø,1;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;

MEM_BLOCK 512,16;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;

MEM_BLOCK 78Ø,1;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1Ø24,1;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1Ø4Ø,1;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1Ø48,1;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1Ø56,1;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1Ø72,1;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1Ø88,1;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;

MEM_BLOCK 11Ø4,1;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;

MEM_BLOCK 112Ø,1;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1136,1;
  ØØØ1 ØØ1Ø;
```

```
END_MEM_BLOCK;

MEM_BLOCK 1536,3;
  ØØØ1 ØØ1Ø;
  ØØØ1 1111;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1568,8;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1584,4;
  ØØØ1 111Ø;
  ØØØ1 ØØ11;
  ØØØ1 ØØ1Ø;
  Ø1Ø1 1Ø11;
END_MEM_BLOCK;

MEM_BLOCK 16ØØ,15;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1616,5;
  ØØØ1 ØØ1Ø;
  Ø111 ØØ1Ø;
  Ø3Ø1 ØØ1Ø;
  1ØØ1 ØØ1Ø;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1632,3;
  ØØØ1 ØØØØ;
  1111 ØØØØ;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;

MEM_BLOCK 1648,3;
  ØØØØ ØØ1Ø;
  111Ø ØØ1Ø;
  ØØØ1 ØØ1Ø;
END_MEM_BLOCK;
```

```
MEM_BLOCK 1843,10;
   0001 0010;
   0001 0010;
   0001 0010;
   0001 0010;
   0001 0010;
   0001 0010;
   0001 0010;
   0001 0010;
   0001 0010;
   0001 0010;
END_MEM_BLOCK;

END.
```

```
FILE_TYPE = MEMORY_CONTENTS;
BIT_RANGE = 7..0;

MEM_BLOCK 0,1;
  0110 1010;
END_MEM_BLOCK;

MEM_BLOCK 512,16;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
END_MEM_BLOCK;

MEM_BLOCK 780,1;
  0110 1010;
END_MEM_BLOCK;

MEM_BLOCK 1024,1;
  0110 1010;
END_MEM_BLOCK;

MEM_BLOCK 1040,1;
  0110 1010;
END_MEM_BLOCK;

MEM_BLOCK 1048,1;
  0110 1010;
END_MEM_BLOCK;

MEM_BLOCK 1056,1;
  0110 1010;
END_MEM_BLOCK;

MEM_BLOCK 1072,1;
  0110 1010;
END_MEM_BLOCK;

MEM_BLOCK 1088,1;
  0110 1010;
END_MEM_BLOCK;

MEM_BLOCK 1104,1;
  0110 1010;
END_MEM_BLOCK;

MEM_BLOCK 1120,1;
  0110 1010;
END_MEM_BLOCK;

MEM_BLOCK 1136,1;
  0110 1010;
```

```
END_MEM_BLOCK;

MEM_BLOCK 1536,3;
  0110 1010;
  1110 1010;
  0010 1010;
END_MEM_BLOCK;

MEM_BLOCK 1568,8;
  0100 1010;
  0101 1010;
  0110 1010;
  0110 0000;
  0110 1000;
  0110 1000;
  0110 1000;
  0010 1000;
END_MEM_BLOCK;

MEM_BLOCK 1584,4;
  0110 1010;
  1110 1010;
  0100 0010;
  0110 1010;
END_MEM_BLOCK;

MEM_BLOCK 1600,15;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1010;
  0110 1001;
  0110 1001;
  0010 1010;
END_MEM_BLOCK;

MEM_BLOCK 1616,5;
  0110 1010;
  0110 1010;
  0110 1000;
  0110 1000;
  0010 1010;
END_MEM_BLOCK;

MEM_BLOCK 1632,3;
  0110 1010;
  0110 1010;
  0010 1010;
END_MEM_BLOCK;

MEM_BLOCK 1648,3;
  0110 1010;
  0110 1010;
  0010 1010;
END_MEM_BLOCK;
```

```
MEM_BLOCK 1843,10;
   0110 1010;
   0110 1010;
   0110 1010;
   0110 1010;
   0110 1010;
   0110 1010;
   0110 1001;
   0110 1001;
   0110 1010;
   0010 1010;
END_MEM_BLOCK;

END.
```

```
FILE_TYPE = MEMORY_CONTENTS;
BIT_RANGE = 7..0;

MEM_BLOCK 0,1;
  1010 0000;
END_MEM_BLOCK;

MEM_BLOCK 512,16;
  1000 0000;
  1000 0000;
  1000 0000;
  1000 0000;
  1000 0000;
  1000 0000;
  1000 0000;
  1000 0000;
  1000 0000;
  1000 0000;
  1000 0000;
  1000 0000;
  1010 0000;
  1000 0000;
END_MEM_BLOCK;

MEM_BLOCK 780,1;
  0110 0000;
END_MEM_BLOCK;

MEM_BLOCK 1024,4;
  1010 0000;
  1010 0000;
  1010 0000;
  1010 0000;
END_MEM_BLOCK;

MEM_BLOCK 1040,1;
  1010 0000;
END_MEM_BLOCK;

MEM_BLOCK 1048,1;
  1010 0000;
END_MEM_BLOCK;

MEM_BLOCK 1056,1;
  1010 0000;
END_MEM_BLOCK;

MEM_BLOCK 1072,1;
  1010 0000;
END_MEM_BLOCK;

MEM_BLOCK 1088,1;
  1010 0000;
END_MEM_BLOCK;

MEM_BLOCK 1104,1;
  1010 0000;
END_MEM_BLOCK;

MEM_BLOCK 1120,1;
  1010 0000;
END_MEM_BLOCK;
```

```
MEM_BLOCK 1136,1;
  1010 0000;
END_MEM_BLOCK;

MEM_BLOCK 1536,3;
  1010 0001;
  1010 0010;
  1000 0000;
END_MEM_BLOCK;

MEM_BLOCK 1568,8;
  0010 0001;
  0010 0010;
  1010 0011;
  1011 0100;
  1010 0101;
  1010 0110;
  1010 0111;
  1000 0000;
END_MEM_BLOCK;

MEM_BLOCK 1584,4;
  1010 0001;
  1010 0010;
  1010 0011;
  1011 0011;
END_MEM_BLOCK;

MEM_BLOCK 1600,15;
  1010 0001;
  1010 0010;
  1010 0011;
  1010 0100;
  1010 0101;
  1010 0110;
  1000 0111;
  1010 1000;
  0110 1001;
  0110 1010;
  0110 1011;
  0110 1100;
  1010 1101;
  1010 1110;
  1010 0000;
END_MEM_BLOCK;

MEM_BLOCK 1616,5;
  0010 0001;
  0010 0010;
  1010 0011;
  1010 0100;
  1000 0000;
END_MEM_BLOCK;

MEM_BLOCK 1632,3;
  1010 0001;
  1010 0010;
  1000 0000;
END_MEM_BLOCK;

MEM_BLOCK 1648,3;
  1010 0001;
  1010 0010;
```

```
   1000 0000;
END_MEM_BLOCK;

MEM_BLOCK 1843,10;
   1001 0100;
   1011 0101;
   0111 0110;
   0111 0111;
   0111 1000;
   0111 1001;
   1011 1010;
   1011 1011;
   0111 1100;
   0110 0000;
END_MEM_BLOCK;

END.
```

APPENDIX D: SCALD SCRIPT FILES

The script files used for the major simulations in the thesis are included in this Appendix. Instructions 010, 101, 011, and Serial Transfer In From External Interstages are shown on pages 163 - 169. The remaining script files are as follows:

```
HIS 10000
WAV 0 1000

MEMPATH (IN5CON CONTROLLER60P .27S291.89P MEM3P)
MEMLOAD prom1.dat
MEMPATH (IN5CON CONTROLLER60P .27S291.88P MEM3P)
MEMLOAD prom2.dat
MEMPATH (IN5CON CONTROLLER60P .27S291.87P MEM3P)
MEMLOAD prom3.dat
MEMPATH (IN5CON CONTROLLER60P .27S291.86P MEM3P)
MEMLOAD prom4.dat


LOADS EXTERNAL REGS FOR SERIAL TO B' & C'

PAUSE

OPEN ENCCHAN*
DEPOSIT 1
OPEN ENBCHAN*
DEPOSIT 1


OPEN SCCHAN<1..0>
DEPOSIT 3
OPEN SBCHAN<1..0>
DEPOSIT 3

OPEN EXTCIN
OPEN EXTBIN

OPEN CCLKIN
OPEN BCLKIN

OPEN CCHAN<15..0>
OPEN CCHAN<31..16>
OPEN BCHAN<15..0>
OPEN BCHAN<31..16>
OPEN INC<31..0>
DEPOSIT D9885A5A
OPEN INB<31..0>
DEPOSIT D88CA5A5

OPEN BUFC<31..0>
DEPOSIT 00000000
OPEN BUFB<31..0>
DEPOSIT 00000000

OPEN INCLKC !C 70-90
OPEN INCLKB !C 60-85


OPEN INCLKCON
DEPOSIT 0
OPEN INCLKBON
DEPOSIT 0

SI C

OPEN INCLKCON
DEPOSIT 1
OPEN INCLKBON
DEPOSIT 1
SI C
```

```
OPEN BUFC<31..Ø>
DEPOSIT FFFFFFFF
OPEN BUFB<31..Ø>
DEPOSIT FFFFFFFF

OPEN SCCHAN<1..Ø>
DEPOSIT Ø
OPEN SBCHAN<1..Ø>
DEPOSIT Ø

SI C

OPEN ENCCHAN*
DEPOSIT Ø
OPEN ENBCHAN*
DEPOSIT Ø

SI C

SI C

OPEN INCLKCON
DEPOSIT Ø
OPEN INCLKBON
DEPOSIT Ø

OPEN RESET*
DEPOSIT Ø
sl 5ØØ

pause

OPEN ENCCHAN*
REMOVE
OPEN ENBCHAN*
REMOVE
OPEN SCCHAN<1..Ø>
REMOVE
OPEN SBCHAN<1..Ø>
REMOVE
OPEN EXTCIN
REMOVE
OPEN EXTBIN
REMOVE
OPEN CCLKIN
REMOVE
OPEN BCLKIN
REMOVE
OPEN CCHAN<15..Ø>
REMOVE
OPEN CCHAN<31..16>
REMOVE
OPEN BCHAN<31..16>
REMOVE
OPEN BCHAN<15..Ø>
REMOVE
OPEN INC<31..Ø>
REMOVE
OPEN INB<31..Ø>
REMOVE
OPEN BUFC<31..Ø>
REMOVE
OPEN BUFB<31..Ø>
```

```
REMOVE
OPEN INCLKC !C 7Ø-9Ø
REMOVE
OPEN INCLKB !C 6Ø-85
REMOVE
OPEN INCLKCON
REMOVE
OPEN INCLKBON
REMOVE
OPEN RESET*
REMOVE

INSTRUCTION 2 LOAD WDT

PAUSE

OPEN  B1
OPEN  SLVTOCPUSPC*
OPEN  CLRSLONREG*
OPEN  Z<15..Ø>
OPEN  Z<31..16>
OPEN  ADD<1Ø..Ø>
OPEN  ADD<6..4>
OPEN LDWDTREG
OPEN LDWDT*
OPEN WDTCOUNT<15..Ø>
OPEN MID<15..Ø>
OPEN WDTSTOP
OPEN WDTCOUNT<15>
OPEN  INBUS<15..Ø>
OPEN  INBUS<31..16>
OPEN  ENBUSLO<1..Ø>
OPEN  ENBUSHI<1..Ø>
OPEN  MBUSHI<15..Ø>

OPEN  BUFCON<15..Ø>
DEPOSIT ØØØØ

OPEN SPC*
DEPOSIT 1

OPEN ST<3..Ø>
DEPOSIT Ø

OPEN BUS<15..Ø>
DEPOSIT FFFF


OPEN CLK

si 15Ø
OPEN RESET*
DEPOSIT 1
SI 17Ø

OPEN ST<3..Ø>
DEPOSIT f
si 1ØØ

OPEN SPC*
DEPOSIT Ø

OPEN BUS<15..Ø>
DEPOSIT 2316
```

```
si 100

OPEN ST<3..0>
DEPOSIT D
OPEN SPC*
DEPOSIT 1
si 100

OPEN BUS<15..0>
DEPOSIT 9016

OPEN SPC*
DEPOSIT 0
si 100

OPEN SPC*
DEPOSIT 1
si 100

OPEN SPC*
DEPOSIT 0
OPEN BUS<15..0>
DEPOSIT 7FC0
si 100

OPEN SPC*
DEPOSIT 1
si 100

OPEN SPC*
DEPOSIT 0
OPEN BUS<15..0>
DEPOSIT FFFF
si 100

OPEN SPC*
DEPOSIT 1
OPEN ST<3..0>
DEPOSIT 0
OPEN BUFCON<15..0>
DEPOSIT FFFF

si 100
SI 200


STARTS SERIAL TRANSFER TO PRIME

PAUSE

OPEN INCLKCON
DEPOSIT 1
OPEN INCLKBON
DEPOSIT 1
OPEN SCCHAN<1..0>
DEPOSIT 2
OPEN SBCHAN<1..0>
DEPOSIT 2


INSTRUCTION 5 LOAD A REG

PAUSE
```

```
SI 1ØØØ
PAUSE

OPEN  BUFCON<15..Ø>
DEPOSIT ØØØØ


OPEN  SPC*
DEPOSIT 1


OPEN ST<3..Ø>
DEPOSIT f
si 1ØØ

OPEN SPC*
DEPOSIT Ø

OPEN BUS<15..Ø>
DEPOSIT 2316
si 1ØØ

OPEN ST<3..Ø>
DEPOSIT D
OPEN SPC*
DEPOSIT 1
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT AA16

OPEN SPC*
DEPOSIT Ø
si 1ØØ

OPEN SPC*
DEPOSIT 1
OPEN ST<3..Ø>
DEPOSIT 9
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT 5A5A
si 1ØØ

OPEN ST<3..Ø>
DEPOSIT Ø
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT 1ØAA
si 2ØØ

OPEN SPC*
DEPOSIT Ø
si 1ØØ

OPEN SPC*
DEPOSIT 1
si 28Ø

OPEN BUFCON<15..Ø>
DEPOSIT FFFF
```

```
OPEN ENA* .
DEPOSIT Ø
si 2ØØ

OPEN BUFCON<15..Ø>
DEPOSIT ØØØØ
OPEN ENA*
DEPOSIT 1

SI 42Ø
PAUSE

OPEN INCLKCON
DEPOSIT Ø
OPEN INCLKBON
DEPOSIT Ø
OPEN SCCHAN<1..Ø>
DEPOSIT Ø
OPEN SBCHAN<1..Ø>
DEPOSIT Ø

SI 3ØØ


INSTRUCTION 3 VOTE

PAUSE

OPEN VOTERIN<2..Ø>
OPEN VOTEROUT<2..Ø>
OPEN PS<3..Ø>
OPEN NS<3..Ø>
OPEN GENCOUNT<7..Ø>
OPEN STOPGENCNT
OPEN LDGENTIMER*

OPEN ST<3..Ø>
DEPOSIT f
si 1ØØ

OPEN SPC*
DEPOSIT Ø

OPEN BUS<15..Ø>
DEPOSIT 2316
si 1ØØ

OPEN ST<3..Ø>
DEPOSIT D
OPEN SPC*
DEPOSIT 1
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT 9816

OPEN SPC*
DEPOSIT Ø
si 1ØØ

OPEN SPC*
DEPOSIT 1
si 1ØØ
```

```
OPEN SPC*
DEPOSIT Ø
OPEN BUS<15..Ø>
DEPOSIT 7FCØ
si 1ØØ

OPEN SPC*
DEPOSIT 1
si 1ØØ

OPEN SPC*
DEPOSIT Ø
OPEN BUS<15..Ø>
DEPOSIT FFFF
si 1ØØ

OPEN SPC*
DEPOSIT 1
OPEN ST<3..Ø>
DEPOSIT Ø
OPEN BUFCON<15..Ø>
DEPOSIT FFFF
si 1ØØ
SI 2ØØ
```

```
his 10000
wav 0 1000

OPEN  B1
OPEN  SLVTOCPUSPC*
OPEN  ENA*
OPEN  ENBUSLO<1..0>
OPEN  ENBUSHI<1..0>
OPEN  CLRSLONREG*

OPEN  BUFCON<15..0>
DEPOSIT 0000

OPEN  SALO<1..0>
OPEN  SAHI<1..0>
OPEN  BUFSSR*
OPEN  MBUSHI<15..0>
OPEN  INBUS<15..0>
OPEN  INBUS<31..16>
OPEN  Z<15..0>
OPEN  Z<31..16>

OPEN  ADD<10..0>
OPEN  ADD<6..4>

OPEN  BUFCON<15..0>
DEPOSIT 0000

OPEN  SPC*
DEPOSIT 1

OPEN  ST<3..0>
DEPOSIT 0

OPEN  BUS<15..0>\I
DEPOSIT ffff

OPEN  RESET*
DEPOSIT 0

OPEN  PHI1 !C 0-40

MEMPATH (IN5CON CONTROLLER60P .27S291.89P MEM3P)
MEMLOAD prom1.dat
MEMPATH (IN5CON CONTROLLER60P .27S291.88P MEM3P)
MEMLOAD prom2.dat
MEMPATH (IN5CON CONTROLLER60P .27S291.87P MEM3P)
MEMLOAD prom3.dat
MEMPATH (IN5CON CONTROLLER60P .27S291.86P MEM3P)
MEMLOAD prom4.dat

PAUSE

si 150

OPEN RESET*
DEPOSIT 1
si 170

OPEN ST<3..0>
DEPOSIT f
si 100

OPEN SPC*
```

```
DEPOSIT Ø

OPEN BUS<15..Ø>
DEPOSIT 2316
si 1ØØ

OPEN ST<3..Ø>
DEPOSIT D
OPEN SPC*
DEPOSIT 1
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT AA16

OPEN SPC*
DEPOSIT Ø
si 1ØØ

OPEN SPC*
DEPOSIT 1
OPEN ST<3..Ø>
DEPOSIT 9
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT 139F
si 1ØØ

OPEN ST<3..Ø>
DEPOSIT Ø
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT ABCD
si 2ØØ

OPEN SPC*
DEPOSIT Ø
OPEN  BUS<15..Ø>
DEPOSIT 3333
si 1ØØ

OPEN SPC*
DEPOSIT 1


si 28Ø

OPEN BUFCON<15..Ø>
DEPOSIT FFFF
OPEN ENA*
DEPOSIT Ø
Sl 1ØØ
DEPOSIT 1
SI 1ØØ

OPEN SAHI<1..Ø>
remove
OPEN SALO<1..Ø>
remove
si 2ØØ

INSTRUCTION 4 LOAD A TO CPU
```

```
OPEN  MBUSHI<15..Ø>

OPEN  BUFCON<15..Ø>
DEPOSIT ØØØØ


OPEN  SPC*
DEPOSIT 1

OPEN  ST<3..Ø>
DEPOSIT Ø

OPEN  BUS<15..Ø>\I
DEPOSIT ffff


si 15Ø

si 17Ø

OPEN ST<3..Ø>
DEPOSIT f
si 1ØØ

OPEN SPC*
DEPOSIT Ø

OPEN BUS<15..Ø>
DEPOSIT 2316
si 1ØØ

OPEN ST<3..Ø>
DEPOSIT D
OPEN SPC*
DEPOSIT 1
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT A2D6

OPEN SPC*
DEPOSIT Ø
si 1ØØ

OPEN SPC*
DEPOSIT 1
OPEN ST<3..Ø>
DEPOSIT 9
OPEN  BUFCON<15..Ø>
DEPOSIT FFFF
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT BBBB
si 1ØØ

OPEN ST<3..Ø>
DEPOSIT Ø
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT CCCC
```

```
si 2ØØ

OPEN SPC*
DEPOSIT Ø
si 1ØØ

OPEN SPC*
DEPOSIT 1
si 68Ø
si 5ØØ
```

```
his 10000
wav 0 1000


OPEN  INBUS<15..0>
OPEN  INBUS<31..16>
OPEN  B1
OPEN  SLVTOCPUSPC*
OPEN CLRSLONREG*
OPEN  ENC*
OPEN  ENB*
OPEN  ENA*
OPEN  ENBUSLO<1..0>
OPEN  ENBUSHI<1..0>
OPEN  SC<1..0>
OPEN  SB<1..0>
OPEN  SALO<1..0>
OPEN  SAHI<1..0>
OPEN  CLK
OPEN  Z<15..0>
OPEN  Z<31..16>

OPEN  MBUSHI<15..0>

OPEN  BUFCON<15..0>
DEPOSIT 0000
OPEN  BUS<15..0>\I
DEPOSIT ffff

OPEN  ADD<10..0>
OPEN  ADD<6..4>

OPEN  SPC*
DEPOSIT 1

OPEN  ST<3..0>
DEPOSIT 0


OPEN  RESET*
DEPOSIT 0

MEMPATH (IN5CON CONTROLLER60P .27S291.89P MEM3P)
MEMLOAD prom1.dat
MEMPATH (IN5CON CONTROLLER60P .27S291.88P MEM3P)
MEMLOAD prom2.dat
MEMPATH (IN5CON CONTROLLER60P .27S291.87P MEM3P)
MEMLOAD prom3.dat
MEMPATH (IN5CON CONTROLLER60P .27S291.86P MEM3P)
MEMLOAD prom4.dat


si 150

OPEN RESET*
DEPOSIT 1
si 170

OPEN ST<3..0>
DEPOSIT f
si 100

OPEN SPC*
DEPOSIT 0
```

```
OPEN BUS<15..Ø>
DEPOSIT 2316
si 1ØØ

OPEN ST<3..Ø>
DEPOSIT D
OPEN SPC*
DEPOSIT 1
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT AA16

OPEN SPC*
DEPOSIT Ø
si 1ØØ

OPEN SPC*
DEPOSIT 1
OPEN ST<3..Ø>
DEPOSIT 9
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT 4567
si 1ØØ

OPEN ST<3..Ø>
DEPOSIT Ø
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT Ø123
si 2ØØ

OPEN BUS
REMOVE

OPEN BUFCON
REMOVE

OPEN MBUSHI
REMOVE

OPEN ENBUSHI
REMOVE

OPEN ENBUSLO
REMOVE


PAUSE

OPEN ENA*
DEPOSIT Ø
SI 1ØØ
OPEN ENA*
DEPOSIT 1
si 1ØØ

OPEN SPC*
DEPOSIT 1
si 1ØØ
```

```
OPEN ST<3..Ø>
DEPOSIT f
si 1ØØ

OPEN SPC*
DEPOSIT Ø

OPEN BUS<15..Ø>
DEPOSIT 2316
si 1ØØ

OPEN ST<3..Ø>
DEPOSIT D
OPEN SPC*
DEPOSIT 1
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT 8216

OPEN SPC*
DEPOSIT Ø
si 1ØØ

OPEN SPC*
DEPOSIT 1
OPEN ST<3..Ø>
DEPOSIT 9
si IØØ

OPEN BUS<15..Ø>
DEPOSIT I39F
si 1ØØ

OPEN ST<3..Ø>
DEPOSIT Ø
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT ABCD
si 2ØØ

OPEN SPC*
DEPOSIT Ø

OPEN ENB*
DEPOSIT Ø
si 1ØØ

OPEN SPC*
DEPOSIT 1
OPEN ENB*
DEPOSIT I
OPEN ENC*
DEPOSIT Ø
SI 1ØØ
DEPOSIT 1
SI C
```

```
his 10000.
wav 0 1000


OPEN  INBUS<15..0>
OPEN  1NBUS<31..16>
OPEN  B1
OPEN  SLVTOCPUSPC*
OPEN CLRSLONREG*
OPEN  ENC*
DEPOS1T 1
OPEN  ENB*
DEPOS1T 1
OPEN  ENA*
DEPOSIT 1
OPEN  ENBUSLO<1..0>
DEPOS1T 0
OPEN  ENBUSH1<1..0>
DEPOS1T 0
OPEN  SC<1..0>
OPEN  SB<1..0>
OPEN  SALO<1..0>
OPEN  SAH1<1..0>
OPEN  CLK
OPEN  Z<15..0>
OPEN  Z<31..16>

OPEN  MBUSH1<15..0>

OPEN  BUFCON<15..0>
DEPOS1T 0000
OPEN  BUS<15..0>\I
DEPOS1T face

OPEN  ADD<10..0>
OPEN  ADD<6..4>

OPEN  SPC*
DEPOS1T 1

OPEN  ST<3..0>
DEPOS1T 0


OPEN  RESET*
DEPOSIT 0

MEMPATH (1N5CON CONTROLLER60P .27S291.89P MEM3P)
MEMLOAD prom1.dat
MEMPATH (1N5CON CONTROLLER60P .27S291.88P MEM3P)
MEMLOAD prom2.dat
MEMPATH (1N5CON CONTROLLER60P .27S291.87P MEM3P)
MEMLOAD prom3.dat
MEMPATH (1N5CON CONTROLLER60P .27S291.86P MEM3P)
MEMLOAD prom4.dat


open enpb*
deposit 1
open enpc*
deposit 1
pause
si 130
si c
```

```
open enbuslo
deposit Ø
open enbushi
deposit Ø
si 1ØØ
resume
open sb
deposit 3
si 2ØØ
open bus
deposit cafe
open sb
deposit Ø
open sc
deposit 3
si 2ØØ

deposit Ø
open bus
deposit ffff
open enbushi
deposit 2
open enbuslo
deposit 2
open enpb*
remove

open enpc*
remove


pause
open phil !c Ø-4Ø
open clk
remove

open enbuslo
remove

open enbushi
remove

open mbushi
remove

open inbus<15..Ø>
remove

open inbus
open inbus<31..16>
remove

resume

si 2ØØ

open enb*
deposit Ø
si 1ØØ
deposit 1
open enc*
deposit Ø
si 1ØØ
deposit 1
```

```
PAUSE
deposit Ø
deposit 1
si 2ØØ
deposit Ø
si 2ØØ
deposit 1
open enb*
deposit Ø
si 2ØØ
deposit 1
si 7Ø
resume

si 15Ø

OPEN RESET*
DEPOSIT 1
si 17Ø



INSTRUCTION 6, C TO A

OPEN ST<3..Ø>
DEPOSIT f
si 1ØØ

OPEN SPC*
DEPOSIT Ø

OPEN BUS<15..Ø>
DEPOSIT 2316
si 1ØØ

OPEN ST<3..Ø>
DEPOSIT D
OPEN SPC*
DEPOSIT 1
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT f321

OPEN SPC*
DEPOSIT Ø
si 1ØØ

OPEN SPC*
DEPOSIT 1
OPEN ST<3..Ø>
DEPOSIT 9
si 1ØØ

OPEN BUS<15..Ø>
DEPOSIT 1111
si 1ØØ

OPEN ST<3..Ø>
DEPOSIT Ø
```

```
si 100

OPEN BUS<15..0>
DEPOSIT 2222

OPEN ST<3..0>
DEPOSIT 1
SI 100


SI 400

pause
open ena*
dep
 0
si 200
deposit 1
si c
resume


INSTRUCTION 7, B TO A

OPEN ST<3..0>
DEPOSIT f
si 100

OPEN SPC*
DEPOSIT 0

OPEN BUS<15..0>
DEPOSIT 2316
si 100

OPEN ST<3..0>
DEPOSIT D
OPEN SPC*
DEPOSIT 1
si 100

OPEN BUS<15..0>
DEPOSIT 3bad

OPEN SPC*
DEPOSIT 0
si 100

OPEN SPC*
DEPOSIT 1
OPEN ST<3..0>
DEPOSIT 9
si 100

OPEN BUS<15..0>
DEPOSIT 3333
si 100

OPEN ST<3..0>
DEPOSIT 0
si 100

OPEN BUS<15..0>
DEPOSIT 4444
```

```
OPEN ST<3..Ø>
DEPOSIT 1
SI 1ØØ


SI 4ØØ


open ena*
deposit Ø
si 2ØØ
deposit 1
si 1ØØ
plot 13ØØ 512Ø in67plot.dat
```

LIST OF REFERENCES

1. Szelar, Ken J., and others, Digital Fly By Wire Flight Control Validation Experience, NASA Technical Manual 72860, Dec 78.

2. Abbott, Larry W., Operational Characteristics of the Dispersed Sensor Processor Mesh, IEEE/AIAA 5th Digital Avionics Systems Conference, Seattle, WA, Oct 31 - Nov 3 1983.

3. Abbott, Larry W., Test Experience on an Ultrareliable Computer Communication Network, IEEE/AIAA 6th Digital Avionics Systems Conference, Baltimore, MA, Dec 3-6 1984.

4. Abbott, Larry W., A Synergistic Fault Tolerant Computer Design for an N-Version Programming Element, Naval Postgraduate School, Monterey, CA, Nov 1984.

5. Siewiorek, Daniel P., and McCluskey, Edward J., "An Iterative Cell Switch Design for Hybrid Redundancy", IEEE Transactions on Computers, Vol. C-22, No. 3, March, 1973

6. Smith, T. Basil, Fault Tolerant Processor Concepts and Operation, CSDL-P-1727, Charles Stark Draper Laboratory Cambridge, Massachusetts, May 1, 1983.

7. Series 32000™ National Semiconductor Corporation, Databook, Santa Clara, CA, 1984.

8. Mano, M. M., Digital Logic and Computer Design, pp. 102 - 112, Prentice-Hall, Inc., 1979.

9. Draft 8.0 of IEEE Task P754, "A Proposed Standard for Binary Floating Point Arithmetic", IEEE Computer, March 1981.

10. Fletcher, William I., An Engineering Approach to Digital Design, pp. 743-760, Prentice-Hall, Inc., 1980.

11. Fairchild TTL Databook, Fairchild Camera & Instrument Company, Mountainview, CA, 1978

12. Fairchild FAST Databook, Fairchild Camera & Instrument Company, South Portland, Maine, 1982

BIBLIOGRAPHY

13th Annual International Symposium on Fault Tolerant Computing: Digest of Papers, IEEE Computer Society, June 28-30, 1983, Milano, Italy.

14th Annual International Symposium on Fault Tolerant Computing: Digest of Papers, IEEE Computer Society, June 20-22, 1984, Kissimmee, Florida.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
2.	Chairman, Department of Electrical and Computer Engineering (Code 62) Naval Postgraduate School Monterey, California 93943-5000	2
3.	Professor Larry Abbott, Code 62At Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	8
4.	Captain William J. Luk, USA 5313 Montclair Drive Raleigh, North Carolina 27609	1
5.	Captain Virgil K. Spurlock, USA 1037 Garvin Place Louisville, Kentucky 40203	5
6.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2

217602

Thesis
S66891 Spurlock
c.1 Design and simulation of an ultra reliable fault tolerant computing system voter and interstage.